

AD-A188 832

A PRODUCTION-QUALITY UNIX VERY HIGH SPEED INTEGRATED
CIRCUIT (VHSIC) HARD. (U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. R M BRATTON

1/2

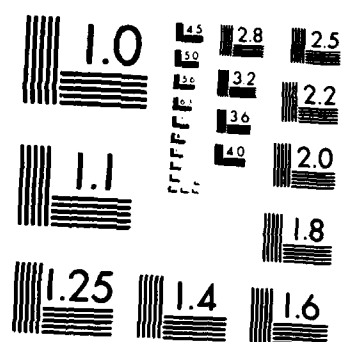
UNCLASSIFIED

DEC 87 AFIT/GCS/MA/87D-1

F/G 12/5

ML

A 10x10 grid of squares, with the top-left square missing, forming a shape resembling a staircase or a corner.



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

DTIC FILE COPY

1

AD-A188 832



A PRODUCTION-QUALITY UNIX
VERY HIGH SPEED INTEGRATED CIRCUIT (VHSIC)
HARDWARE DESCRIPTION LANGUAGE (VHDL)
SUBSET ANALYZER

THESIS

Randolph M. Bratton
Captain, USAF

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

DTIC
ELECTE
FEB 1 0 1988
S D E

Wright-Patterson Air Force Base, Ohio

This document has been approved
for public release and sale; its
distribution is unlimited.

88 2 4 046

AFIT/GCS/MA/87D-1

A PRODUCTION-QUALITY UNIX
VERY HIGH SPEED INTEGRATED CIRCUIT (VHSIC)
HARDWARE DESCRIPTION LANGUAGE (VHDL)
SUBSET ANALYZER

THESIS

Randolph M. Bratton
Captain, USAF

AFIT/GCS/MA/87D-1

DTIC
ELECTE
S
E

Approved for public release; distribution unlimited

**A PRODUCTION-QUALITY UNIX
VERY HIGH SPEED INTEGRATED CIRCUIT (VHSIC)
HARDWARE DESCRIPTION LANGUAGE (VHDL)
SUBSET ANALYZER**

THESIS

**Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Science**

**Randolph M. Bratton, B.S., B.S.E.T.
Captain, USAF**



Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail number Special
A-1	

December 1987

Approved for public release; distribution unlimited

Acknowledgements

I am deeply indebted to the many persons who aided my research efforts presented in this report. Of these, there are a few to whom I would like to express a special thanks. First, thanks go to my thesis readers: Maj Joseph W. DeGroat, who also worked actively in the integration of the AFIT UNIX Analyzer and Simulator; and Capt James W. Howatt, who offered guidance in the areas of software quality and testing. I also appreciate the support from my thesis sponsors, Mr. R. Wallace and Capt J.R. Tomlinson from the Air Force Wright Aeronautical Labs, especially for the use of the VHDL VMS Test Suite. LTC John Dallen, now at the U.S. Military Academy, graciously took the time to send me a copy of his *Patois* research, which was precisely the new approach my research needed. To fellow students and VHDL "gurus" CPT Mike Dukes and 2Lt Harvey Kodama--thanks for a job well done; I think it was worth all the sweat and frustration. To Capt Al Deese, thanks for being a "sounding board" for my ideas, and to both him and his wife, Dale, thanks for your friendship and words of encouragement.

Finally, thanks go to the two persons most responsible for the completion of this work. LtCol Richard R. Gross, my thesis advisor, was always willing to discuss my ideas, to point out possible pitfalls in my writing, and like every

good thesis advisor, to allow me to work through my own problems. Thank you for all your help (even though I still have nightmares about getting back my rough drafts completely covered in red pencil marks). And to my wife, Kay--I couldn't have done it without you. Thanks for all the love and understanding through some difficult times. God bless you!

Randolph M. Bratton

Table of Contents

	Page
Acknowledgments	ii
List of Figures	vii
List of Tables	viii
List of Analyzer Listings	ix
Abstract	x
I. Problem Statement	1.1
Statement of the Problem	1.1
Background	1.1
Scope	1.4
The VHDL Analyzer	1.5
Production-quality	1.7
Production Quality UNIX Analyzer Criteria and Requirements	1.7
The Prototype Analyzer	1.13
Research Approach	1.14
Maximum Expected Gain	1.18
Overview of the Thesis	1.18
II. Literature Review	2.1
Introduction	2.1
UNIX VHDL Analyzers	2.1
Error Recovery	2.2
Error Repair	2.3
Error Recovery	2.4
Intermediate Representation	2.5
Intermediate VHDL Attributed Notation (IVAN)	2.5
Design Data Structure (DDS)	2.7
VHDL Intermediate Access (VIA)	2.10
University of Pittsburgh VHDL	2.12
Other HDLs	2.14
Summary	2.15

III.	System Design	3.1
	Overview of Chapter	3.1
	Find a more efficient Intermediate Representation	3.1
	Document the IR for use by other tools in the AVE	3.10
	Adapt the prototype for use of the new IR	3.10
	Add error handling capabilities to the Analyzer	3.13
	Construct/obtain a test suite	3.14
	Add full language capabilities to the Analyzer	3.14
	Perform Analyzer and system integration tests	3.14
	Document the Results	3.15
	Summary	3.15
IV.	Detailed Design	4.1
	Overview	4.1
	VIA Modification	4.1
	Use of lex and yacc	4.4
	Language Implementation	4.6
	Implementation Examples	4.8
	Behavior Example	4.8
	Structure Example	4.13
	Summary	4.16
V.	Testing and Analysis	5.1
	Introduction	5.1
	Thesis requirements	5.1
	Conformance tests	5.2
	Portability tests	5.3
	Performance tests	5.3
	Integration tests	5.4
	Conformance Testing	5.4
	LRM Conformance	5.4
	Error testing	5.8
	Portability Testing	5.8
	Performance Testing	5.9
	Speed	5.9
	Memory Usage	5.10
	Disk Usage	5.11
	Integration Testing	5.11
	Method	5.11
	Results and Comparisons with VMS VHDL	5.12
	Chapter Summary	5.12
VI.	Conclusions and Recommendations	6.1
	Conclusions	6.1
	Selection of VIA	6.1
	Test Results	6.2

Recommendations for Future Research	6.3
Implement Entire Language	6.3
Add to User Options	6.3
Optimize VIA	6.4
Add Design Library	6.5
Implement other AVE Tools	6.6
Summary	6.6
Appendix A: AFIT VHDL Analyzer Implementation	A.1
Appendix B: VIA Definition	B.1
Appendix C: VHDL Analyzer Test Suite	C.1
Bibliography	Bib.1
Vita	Vita.1

List of Figures

Figure	Page
1.1 AFIT VHDL Environment	1.5
2.1 Parse and Abstract Syntax Trees	2.6
2.2 Signal Assignment Statement Modelled in DDS	2.9
2.3 VHDL Represented in VIA/DDS	2.12
2.4 VIA Record Hierarchy	2.13
3.1 VHDL Source and Resulting AST and C Code	3.8
3.2 Analyzer Design	3.11
4.1 Full-Adder Example	4.3
4.2 VIA for if statement	4.10
4.3 VIA for signal_declaration	4.15

List of Tables

Table	Page
1.1 Implementation of the Prototype Analyzer	1.14
5.1 Results of VMS Test Suite Testing	5.9
5.2 Results of Portability Tests	5.10

List of Analyzer Listings

Listing	Page
5.1 Syntactic Conformance Testing	5.6
5.2 Semantic Error Testing	5.7

Abstract

This paper describes the design and implementation of the Air Force Institute of Technology's (AFIT's) UNIX-based VHDL Analyzer. The purpose of this tool is to facilitate the introduction of VHDL into the academic environment, which may not be able to use the Department of Defense's VMS-based software. This research emphasized two areas: the criteria for a "production-quality" software product and the design of an efficient Intermediate Representation (IR) that serves as an interface between the Analyzer and other tools in the AFIT VHDL Environment (AVE). Background on other UNIX VHDL analyzers, as well as other IRs, was presented. A two-part IR, based on Dallen's *Patois* hardware description language and named the VHDL Intermediate Access (VIA), was designed, and examples were given that illustrate its use. Test results showed that the Analyzer passed over 75% of the conformance tests from the VHDL VMS Analyzer Test Suite and performed well in the areas of compile time, memory usage, and disk usage. Recommendations for future research include adding user options to the Analyzer and implementing a design library for VHDL designs.

**A Production-Quality UNIX
Very High Speed Integrated Circuit (VHSIC)
Hardware Description Language
Subset Analyzer**

I. Problem Statement

Statement of the Problem

The Department of Defense (DoD) has standardized a hardware description language for Very High Speed Integrated Circuits (VHSIC) called the VHSIC Hardware Description Language or VHDL. To gain widespread acceptance in the academic community, the VHDL software tools (analyzers, simulators, and code retargeters) must work in the computer environment many universities now use--UNIX¹. But, as yet, no such UNIX VHDL toolset exists.

Background

To put this problem in perspective, I will briefly describe VHDL: what is it, why is it important, what makes a UNIX VHDL design environment desirable, and what needs must be met to produce such a UNIX VHDL environment. I will also relate current research in VHDL to these needs.

¹UNIX is a trademark of Bell Laboratories.

VHDL, among other hardware description languages, is used to design, document, and validate hardware components (Lieberherr, 1985:55). Its syntax and semantics allow hardware designs to be precisely specified and those designs to be unambiguously transferred among design engineers and organizations (Lipsett and others, 1986:28). But, other hardware description languages, such as ZEUS (Lieberherr, 1985), CONLAN (Piloty and Borriore, 1985), and Computer Design Language (CDL) (Chu, 1972) among others, have been widely used, some being used as early as the 1960s. And in this widespread use lies part of the problem facing the Air Force today.

In February 1985, the Air Force issued its reliability and maintainability (R&M) plan, called "R&M 2000," which is intended to provide greater reliability and maintainability in defense systems (Goodman, 1987:58). Specifically, in contracts for new defense systems, R&M criteria are now given as much weight as cost and performance. One of the main goals of R&M 2000 is the use of VHSIC technology in place of more bulky circuit boards in aircraft avionics systems. For example, one VHSIC board can take the place of several line-replaceable units currently used and, at the same time, offer a speed increase on the order of three to four times. Also, the number of maintenance personnel necessary to maintain and repair the VHSIC boards can be reduced (Goodman, 1987:58). However, such advances in

technology do not happen without a price. "Already in the \$2 to \$5 million range, development costs of advanced ICs [integrated circuits] must be reduced to economically meet future government IC demands" (Dewey and Gadiant, 1986:13). The use of hardware description languages can help reduce these development costs.

As mentioned above, many HDLs exist but none have gained widespread acceptance (Lieberherr, 1985:55; Nash, 1984:18). Because of this, the Department of Defense began a program, VHDL, to standardize a hardware description language for VHSIC systems (Dewey and Gadiant, 1986:12). VHDL is an Ada²-based hardware description language. Its goals are to "address the broad range of descriptive abilities required for advanced electronic system documentation, and to establish a standard for eliminating current diversity in hardware description languages" (Dewey and Gadiant, 1986:13). VHDL is designed to be flexible: the VHSIC design engineer need not be concerned about the design style or technology used (Waxman, 1986:92-93).

VHDL tools delivered under the DoD contract (ASD, 1983) run under the Digital Equipment Corporation (DEC) VAX/VMS (Virtual Address eXtended/Virtual Memory System) operating

² Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

system³ (Deitel, 1984:507-508). This operating system (or environment) is in general use in the commercial world. Universities, given the responsibility to educate their engineering students in the use of HDLs, need a VHDL toolset to run under UNIX (the other major operating system for the VAX and a *de facto* university standard). To meet this need, AFIT and the Air Force Wright Aeronautical Laboratories (AFWAL) decided in 1985 to investigate the "development of a UNIX-based VHDL integrated tool set, which subsequently became known as the AFIT VHDL Environment (AVE)" (Carter and others, 1987:3).

The 1986 prototype AFIT VHDL Environment (see Figure 1.1) consists of a prototype VHDL source code analyzer (Frauenfelder, 1986), a kernel VHDL simulator (Lynch, 1986), a parallel VHDL simulator design (Kamrowski, 1986), and a VHDL-based microcode retargeter design (Decker, 1986). In 1987, two thesis efforts were started to complete the work begun by Frauenfelder and Lynch. Research will continue in the other areas as student interest warrants (Carter and others, 1987:3-6).

Scope

The goal of this project was to produce a production-quality VHDL analyzer, using Frauenfelder's prototype as a

³ This toolset delivered under the DoD contract will be referred to as the VHDL/VMS system in the remainder of this paper.

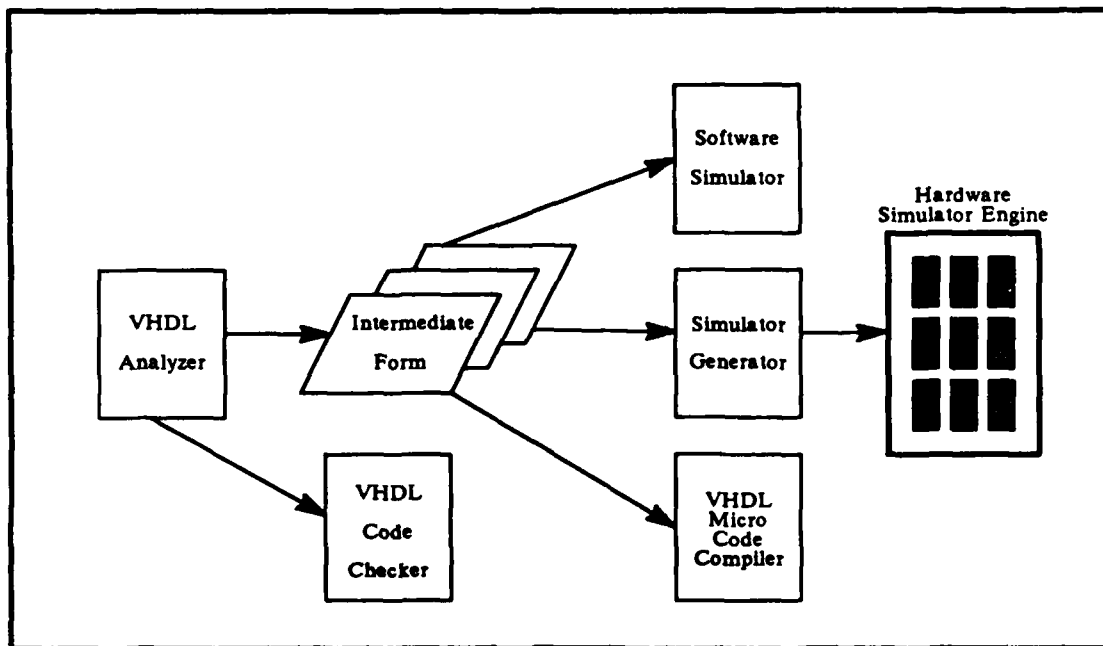


Figure 1.1 AFIT VHDL Environment (Frauenfelder, 1986:1.3)

foundation from which to work. To define this goal more precisely, four steps needed to be completed:

1. Identify what a VHDL analyzer should do.
2. Define what is meant by "production-quality."
3. Apply this definition to the Analyzer project.
4. Define the tasks needed to make the prototype analyzer a production-quality product.

The VHDL Analyzer. The DoD contract specifies the purpose of the VHDL Analyzer. In general, the Analyzer must

(1) check syntax and static semantics of the VHDL source and
(2) organize design and documentation data into a more efficient machine readable form called an Intermediate Representation (IR) (ASD, 1983:11). Specifically, the DoD requires the contract VHDL Analyzer to:

1. Detect all input syntax and static semantic source code errors.
2. Be syntax driven.
3. Be written in Ada.
4. Run on VAX 11/780 under the VMS operating system.
5. Be production-quality (ASD, 1983:11).

Because the IR forms the important interface between the Analyzer and other tools in the VHDL design environment, the DoD requirements for the Intermediate Representation should also be presented. The purpose of the Intermediate Representation is to "allow a lower level description of the VHDL source to serve as an aid in interfacing the VHDL to various design automation tools" (ASD, 1983:11). The IR is meant to solve interfacing issues, not transportability ones, because the VHDL source is the medium for transportability (ASD, 1983:11). An IR for VHDL should (1) be organized in an orderly manner for easy access of any information it contains and (2) be independent of host machine environments and tools (ASD, 1983:11).

Production-quality. The DoD's requirement for production-quality, while certainly necessary, is nonetheless quite general and needs further clarification. In The Mythical Man Month, Brooks asserts that a useful "programming systems product" should have the qualities of:

Generalization. The tool should accept a wide variety of input (correct as well as erroneous) and, if the tool is part of a programming system, it should interface acceptably with various other tools.

Reliability. The tool should be tested thoroughly, by itself and in conjunction with other tools. This also implies the interfaces between each tool must be well-defined and stable.

Maintainability. The tool should be designed in a way that supports testing, modification, and extension. It should be thoroughly documented, both internally and externally.

Economy. Brooks says the tool should use a "prescribed budget of resources" (Brooks, 1975:4-6).

Production Quality UNIX Analyzer Criteria and Requirements. From Brooks' definition, the production-quality UNIX VHDL Analyzer for the AFIT VHDL Environment should meet the following specifications:

Generalization:

1. The Analyzer must run under the latest version UNIX and use those tools commonly found in that programming environment. As discussed above, the projected audience is the academic environment which generally prefers UNIX. Using common UNIX tools, such as the programming language C (Kernighan and Ritchie, 1978) and compiler-compilers like yacc (Johnson, 1978), allows more people to be able to use this product. This requirement precluded the use of Ada or the use of any licensed commercial software, because not all UNIX installations have Ada or have invested in the same types of commercial software tools.

2. The Analyzer must run on several different computer configurations. This showed that the Analyzer is not tied to any specific computer hardware and ensures portability. Specifically, this Analyzer was ported to a VAX 11/780, an ELXSI 6400, and a Sun 2 workstation. The VAX was chosen because it represents the general-purpose class of computer found in many university computer installations. The Elxsi and the Sun are representative of the newer generations of faster and smaller processors that are now being acquired by the academic community.

3. The Analyzer must analyze both correct and incorrect VHDL source. For incorrect VHDL, the Analyzer must recover "gracefully" from any syntax or static semantic

error, with an appropriate error message. It is impossible to ensure that every error be detected, but the Analyzer should not "crash" due to an incorrect input.

Testing:

4. The Analyzer must pass a comprehensive (though not necessarily exhaustive) test suite designed to exercise each VHDL grammar rule and associated program modules as thoroughly as possible. The Analyzer was tested in the same manner and with the same test suite as the VMS version. The VMS VHDL Analyzer Test Plan is documented in (Intermetrics, 1984b) and examples of these tests can be found in Chapter 5 and in Appendix C.

Documentation:

5. The Analyzer, including the intermediate representation, module interfaces, and major data structures, must be well documented. This thesis provides a large part of this documentation by explaining the design decisions made during this project.

Maintenance:

6. The Analyzer should be easily maintainable and extensible. The source code should include in-line documentation and interface specifications for each module or function. In general, it should subscribe to the use of

good software engineering practices, such as information hiding, data abstraction, and module cohesion and coupling (Pressman 1982:Ch. 7).

Interfaces:

7. The Analyzer must produce an Intermediate Representation that provides the necessary information to the other tools in the environment. By "necessary" it is meant that IR should provide at least as much information as contained in the original VHDL, i.e. structure and behavior (Nash and Saunders, 1986), though it should be "distilled" and organized for easy access, according to the DoD requirement. If a tool needs more information than VHDL can supply, then it will not be well-suited for the AVE, unless it does not rely solely on VHDL for input.

8. The Analyzer's IR must be efficient and provide easy access to the information it contains. Comparisons of different IRs for VHDL were used to determine which representation was best for the Analyzer, based on ease of information access and the resulting output data size. Descriptions and examples of the researched IRs can be found in Chapter 2, with the results of those comparisons being presented in Chapter 3.

System Resources:

9. The Analyzer must be able to process VHDL designs within a "reasonable" time period, dependent upon the size of the design. Since many people equate compile times with production-quality, the Analyzer must be efficient in its use of CPU time. The goal for this project was 1000 lines of VHDL source analyzed per CPU minute on an unloaded VAX 11/780 running UNIX bsd 4.2. This goal is the same as for the VMS VHDL implementation (Intermetrics, 1984a:2.2).

10. The Analyzer must conserve the use of main memory and secondary storage in order to be usable on smaller UNIX systems. One goal of this project was that the Analyzer not require more than 640K of main memory for any VHDL description. The resulting output will, of course, be dependent upon the size of the input description, but it should be compact, without sacrificing efficiency. The goal for this requirement was that the IR output should not, on the average, contain more than 100 bytes of data for each VHDL statement. Both these figures are based on the corresponding design goals of the VMS implementation (Intermetrics, 1984a:3.2, 1985b:2.1).

System Integration:

11. After the Analyzer had passed the test suite described under Testing, it had to be tested in conjunction

with the simulator. This further exercised the IR interface and demonstrated the "end-to-end" use of the system, from VHDL input to simulator report output. VHDL designs (such as adders, shifters, etc.) which have a limited number of input/output states were used, so that the resulting simulation was more easily verified.

The scope of this research project did not include several closely related areas, such as:

1. *Optimizing the IR format output.* Optimization of the Intermediate Representation, like object code optimization for software compilers, involves several code-improving translations (Aho, Sethi, and Ullman, 1986:Ch. 10; Barrett and Couch, 1979:Ch. 11). During this project, most of the research was directed towards increasing the capability of the Analyzer. Once the IR had been enhanced and modified, it needed to remain static so that concurrent VHDL projects would have a basis from which to work.

2. *Rewriting the Analyzer to conform to the upcoming IEEE standard VHDL.* VHDL has been proposed as an IEEE standard hardware description language, but with modifications to Version 7.2 (CAD Language Systems, 1986). Because the primary interests in this research are Version 7.2 and the comparison between the 7.2 versions of the VMS and UNIX Analyzers, a separate IEEE VHDL Analyzer was not a product of this project.

The Prototype Analyzer

Because it served as a basis for the production-quality Analyzer, the prototype analyzer was evaluated against the project requirements. This evaluation showed the areas that met the requirements, as well as those needing change or improvement.

Generalization: The prototype runs under UNIX on both a VAX 11/780 and an ELXSI 6400. It is written in C and uses the UNIX tools *lex* (Lesk and Schmidt, 1978) and *yacc* (Johnson, 1978) for lexical and syntactic analysis. It does not include error recovery for syntax errors.

Testing: Testing (and the prototype test suite) was limited to those constructs implemented in the prototype, mainly declarations. (See Table 1.1.)

Documentation: Documentation for the prototype consists of Frauenfelder's thesis (Frauenfelder, 1986) and a technical report detailing the functions of each module in the prototype.

Interfaces: The Intermediate Representation of VHDL produced by the prototype is modeled after Design Data Structure (Knapp and Parker, 1984) and is called the VHDL Intermediate Access (VIA). The VIA will be discussed in more detail in Chapter 2.

Maintenance: The prototype is well modularized and commented. As mentioned above, a maintenance document describes each of the modules functions and expected inputs.

System Resources: The prototype can analyze 1000 lines of VHDL source in under 3 minutes CPU time, however, the total execution time was over 30 minutes. The resulting VIA output file was over 3 Megabytes in size.

Table 1.1 Implementation of the Prototype Analyzer
(Frauenfelder, 1986:A.1-3)

Language Subset	Level of Implementation
1. Design Entities	Implemented except for port lists.
2. Context Clauses	Implemented.
3. Declarations	Implemented except for interface lists and port lists.
4. Expressions	Not Implemented.
5. Sequential Statements	Not Implemented.
6. Concurrent Statements	Not Implemented.
7. Configurations	Implemented.
8. Subprograms	Implemented except for parameter lists.
9. Other	(Not applicable)

System Integration: The prototype has not been integrated with other tools in the AVE.

Research Approach

To extend the prototype to meet the Analyzer requirements, based on the above evaluation, the following sequence of tasks had to be accomplished:

1. Find ways to make the prototype's Intermediate Representation more efficient or find a more efficient IR. The prototype's IR, called the VHDL Intermediate Access

(VIA) (Frauenfelder, 1986:3.16), did not meet the design goal of 100 bytes per VHDL statement, even after some minor modification to reduce its size. Also, VIA's use of variable length records and complex bindings makes it harder to process and, therefore, requires a more complex internal data structure than a representation with a fixed record length and a simpler linked-list or tree structure. These disadvantages of VIA prompted research into other possible IRs, the results of which are presented in Chapter 2.

2. *Document the IR for use by other tools in the AVE.*

Whatever the form of the IR, it must be well documented because it serves as the interface between the VHDL source and the AVE tools.

3. *Adapt the prototype for use of the new IR.* After selection of the IR, the prototype was modified to generate this new IR while keeping the same level of capabilities as the original prototype. Comparisons were then made to ensure the efficiency of the new prototype had not degraded with the modification.

4. *Add error handling capabilities to the Analyzer.*

Error recovery, and understandable error messages, are vital parts of the human/computer interface (Kantorowitz and Laor, 1986:627-629) and allow the tool to accept a wider range of input, i.e. erroneous as well as correct. Error recovery was quickly added to the prototype so that it could continue

to be used early-on in AFIT classes as a VHDL syntax and semantic checker, thus providing feedback on the correctness of the Analyzer's construction throughout the thesis process.

5. *Construct/obtain a test suite for the Analyzer which tests for syntax and static semantic conformance to VHDL Version 7.2. Construct/obtain a test suite that tests the Analyzer's performance.* As mentioned before, the Intermetrics VHDL Analyzer test suite (Intermetrics, 1984b) was used to test for conformance to VHDL Version 7.2. Research by Dukes into the performance of VHDL design environments resulted in a set of VHDL designs used to test system performance and integration (Dukes, 1987). Finally, VHDL designs for common TTL circuits (written by graduate students studying VHDL) were also used in testing system performance and integration, as well as error handling.

6. *Add full language capabilities to the Analyzer.* Following the completion of the VIA modification, the Analyzer was modified to accept the full VHDL language and perform semantic analysis on a major subset of the language. VHDL, like Ada, is a large language with many capabilities. Concurrent work at AFIT by Dukes showed that many of VHDL's features are not absolutely necessary in order to analyze and simulate a large class of circuit designs, even if the designer chooses any of the programming methods offered by

VHDL: dataflow implementation, structural implementation, or procedural implementation (Dukes, 1987). For example, components may be declared and instantiated using the *configuration body declaration* or by using *component declarations* and *instantiations*. By using the second choice, a complete design can be implemented using only *entity* and *architecture* declarations. The implementation of *configuration body* declaration and other alternative methods was deferred so that more effort could be devoted to developing those portions of the language that provided the "most return" on the time spent in terms of being able to simulate a VHDL design.

Frauenfelder divided the VHDL into nine subsets (Table 1.1). What remained of Analyzer development, at a minimum, was to complete the semantic analysis of the language subsets that had not been implemented. In order to maximize the utility and therefore the potential for feedback of the Analyzer while it was still in its maturing phases, the *expression*, *sequential statement*, and *concurrent statement* subsets were implemented first. Each subset was designed, implemented, and tested before proceeding to the next, thereby increasing the capabilities of and feedback from the Analyzer--from source VHDL analysis to VIA output--as each subset was completed.

7. *Perform Analyzer and system integration tests.*

After completion of each subset, testing ensured that the Analyzer produced the correct output for valid VHDL input and produced the correct error message for invalid VHDL input. In this manner, the addition of new language constructs that introduced errors in the existing implementation was quickly discovered and corrected.

8. *Document the Results.* The major design decisions, results, and conclusions are documented in this thesis.

Maximum Expected Gain

It is vital to the future growth of VHSIC technology that universities, as well as industry, become involved in VHSIC research. Universities provide both the Government and industry with well-trained men and women, who are able to design, implement, and manage new VHSIC research projects. The DoD VMS-based VHDL Analyzer will help standardize VHDL in the industrial community, but universities also need the tools to teach tomorrow's engineers and managers. A UNIX-based VHDL Analyzer is an important (and necessary) first step in the insertion of VHSIC technology into the academic community.

Overview of the Thesis

This thesis is presented in six chapters. Chapter 2 reports the results of a literature survey of previous

research on VHDL, Intermediate Representations, and error recovery; it forms the foundation for the system design presented in Chapter 3. Chapter 4 elaborates on the system design, giving the details of the modified VIA format and the language implementation. The testing procedures and analysis of the system design are presented in Chapter 5. Chapter 6 summarizes the findings of this thesis and offers suggestions for future research in the area of VHDL tools.

II. Literature Review

Introduction

Purpose for Literature Review.

A philosopher of imposing stature doesn't think in a vacuum. Even his most abstract ideas are, to some extent, conditioned by what is or is not known in the time when he lives.

--Alfred North Whitehead (1861-1947)

Research, then, depends on the previous knowledge of others. This chapter describes the result of a literature review undertaken:

1. To discover if other VHDL/UNIX research work at other institutions was being conducted. Specifically, I suspected the prototype intermediate representation, VIA, was not optimal for use with a wide variety of tools in the AVE. How were other researchers handling the problem of choosing a VHDL IR? (These results will be presented in the section discussing Intermediate Representations.)
2. To survey existing error recovery techniques, since error recovery, not included in the prototype, must be a capability of the AVE Analyzer.
3. To uncover research with hardware description languages (and, where applicable, software programming languages) that would support the system design decisions that must be made to satisfy the project requirements.

UNIX VHDL Analyzers

Currently, the only other UNIX VHDL research being conducted is at the University of Pittsburgh, under the

direction of Levitan (Levitan, 1987). His design of a "stand-alone" VHDL analyzer/simulator is based on Fraunfelder's prototype, but with several differences. First, he uses C source code as the Intermediate Representation for the analyzer. (This choice is further discussed in the section "Intermediate Representations" in this chapter.) Second, this system is not being initially designed with other tools in mind, as were the prototype and the Analyzer described in this paper.

Error Recovery

One of the requirements for the Analyzer is that it detect errors and recover from those errors, i.e., it must not "crash." A good error handler needs to perform the following functions:

1. It should report the presence of errors clearly and accurately.
2. It should recover from each error quickly enough to be able to detect subsequent errors.
3. It should not significantly slow down the processing of a correct program (Aho, Sethi, and Ullman, 1986:161).

Several methods of error handling exist. Tremblay and Sorenson classify error handling techniques into the categories of acceptable and unacceptable responses to an error. Acceptable responses are reporting the error and

either *recovering* or *repairing* the error¹. Unacceptable responses are not reporting the error and either crashing, looping, or producing an incorrect object program. To report the first error and halt is also unacceptable (Tremblay and Sorenson, 1985:183-185). Since this project required that the Analyzer respond acceptably to errors, either error recovery or error repair were deemed potentially useful techniques.

Error Repair. Error repair modifies the source program to provide subsequent parts of the compiler with a syntactically valid input. This modification takes place by inserting or deleting source text and can be accomplished without modifying text which has already been translated. Several methods of error repair exist: *ad hoc* repair, syntax-directed repair, context-sensitive repair, and spelling repair. *Ad hoc* repair calls special error handling routines which are capable of repairing various classes of common errors. Conway and Wilcox (1973) used this type of error repair in their PL/C compiler. Syntax-directed repair, used by Holt and Barnard (1976), tries to construct a valid syntax tree by inserting a terminal symbol into the source text when one is expected and not found. If one of several terminal symbols can be inserted, an "insertion-cost

¹ Another acceptable response is to report the error and correct the error to what the programmer originally wanted. Even with the current advances in artificial intelligence, this is still not possible (Tremblay and Sorenson, 1985:185; Horning, 1976b:533).

vector" is associated with each terminal symbol and the algorithm attempts to minimize the cost (Tremblay and Sorenson, 1985:201-202). Context-sensitive repair ensures that "repaired" operands have the correct attributes for a particular context. Again, default or universal values are replaced in the source text, similarly to syntax-directed repair (Tremblay and Sorenson, 1985:202-203). A spelling repair algorithm by Morgan (1970) attempts to correct simple typing mistakes causing an identifier to be mistaken for a keyword or vice versa. The types of mistakes Morgan's algorithm corrects are: one symbol changed, inserted, or deleted and two symbols transposed. This algorithm is discussed in more detail in (Tremblay and Sorenson, 1985:203-205).

Error Recovery. There exist several means to achieve error recovery in a compiler or analyzer. The panic mode of error recovery is the simplest to implement, but has the decided disadvantage of throwing away source input that has not been checked for validity (Tremblay and Sorenson, 1985:199). Aho and Johnson propose the use of error tokens (in grammars that define languages such as C) as a means of error recovery. When the parser detects an error in a production containing an error token as a terminal symbol, it replaces the current input with error and reports the error. The parsing stack is then searched for a state that can follow error and parsing continues (Aho and Johnson, 1974; Tremblay and Sorenson, 1985:196; Horning, 1976b:537).

The main disadvantages of this method are that it increases the size and complexity of the grammar and it may make the grammar ambiguous (Horning, 1976b:537). Also, if error tokens are not properly specified, then this method can behave similar to panic mode recovery. On the other hand, the use of error tokens with the UNIX tool yacc is fully documented in (Schreiner and Friedman, 1985:Ch 4). These two facts are important design criteria in deciding whether to implement error recovery rather than error repair.

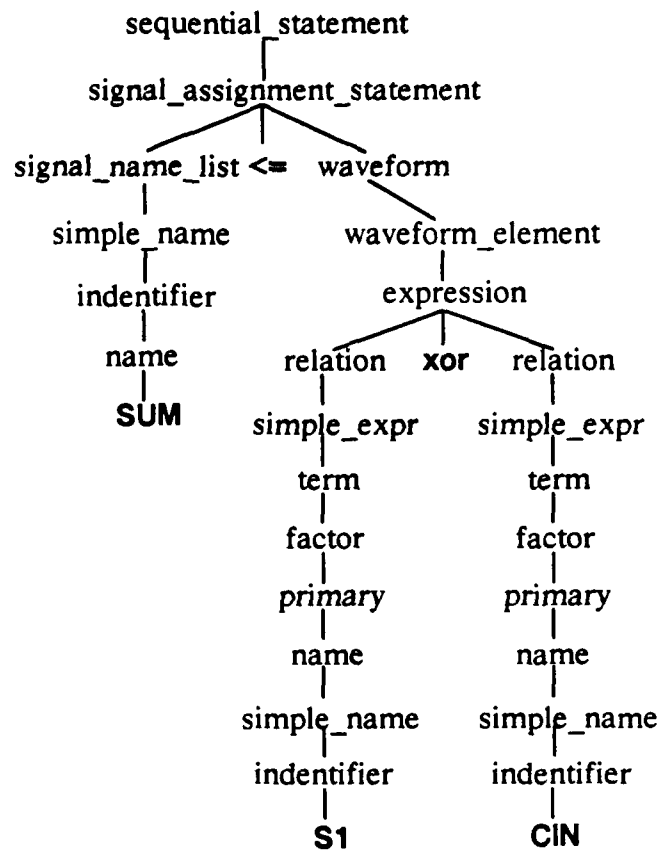
Intermediate Representation

Currently, there are three Intermediate Representations being used with VHDL: Intermediate VHDL Attributed Notation (IVAN) (used in the VMS version), VIA (used with the prototype), and the language C (used with the Pittsburgh version). These were researched first, along with the model for VIA, Design Data Structure (Knapp and Parker, 1984). Rather than restrict the survey to only VHDL implementations, other hardware description languages were studied to determine if any other particular IRs were in frequent use.

Intermediate VHDL Attributed Notation (IVAN) (Gilman, 1986:46). IVAN is an annotated form first produced as an abstract syntax tree (AST) by the lexical and syntactical phases of the VHDL/VMS Analyzer. An AST is a tree where each nonleaf represents an operator and each leaf represents an operand. For example, the AST for a VHDL *signal_assignment* statement is shown in Figure 2.1, along

VHDL Source: **SUM <= S1 xor CIN ;**

Parse Tree:



Abstract Syntax Tree: signal_assignment_statement

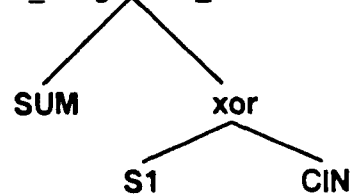


Figure 2.1 Parse and Abstract Syntax Trees

with the more specific parse tree. Later phases of analysis "decorate" the AST with semantic attributes. Thus IVAN is to VHDL as DIANA (Descriptive Intermediate Attributed Notation for Ada) (Evans and others, 1983) is to Ada. In its use with VHDL, IVAN's advantage is that it preserves the exact structure of the original VHDL source. Its main disadvantages are that it, like DIANA, is not a compact representation of the input VHDL and, because it mirrors the original VHDL source so closely, accessing IVAN's information requires the use of a database manager.

Design Data Structure (DDS). DDS forms the basis for a design library database that supports the interactive retrieval for an expert synthesis system (Knapp and Parker, 1984:4). The smallest unit represented in DDS is a *component*, with each component represented in four abstract subspaces: dataflow subspace, timing subspace, structure subspace, and physical subspace. Each subspace or view forms a tree, and complex relationships can exist among the subspaces. Included in the respective subspaces are values for the following entities (Knapp and Parker, 1984:10; Frauenfelder, 1986:2.6-2.11):

dataflow: nodes and values

sequencing: points and ranges

structure: modules and carriers

physical: blocks and nets

An example illuminates DDS capabilities. In VHDL, a *signal* is an object which can be used to connect design components together and can, over the course of time, acquire a series of values. It assumes these values through a *signal assignment statement*, such as the one that follows:

```
SIG <= '1' after 5 ns ;
```

Translated, this statement means that the *signal* "SIG" will get the value TRUE (binary '1') after 5 *nanoseconds* ("ns") have elapsed. In the DDS model, "SIG" is a *carrier*, "<=" is a *node*, '1' is a *value*, and "after 5 ns" represents a *range*. The DDS representation is shown in Figure 2.2. From this representation, one sees the division of this simple statement into three of the four subspaces. *Bindings* exist among the subspaces, shown by the dotted lines connecting those subspaces. As can be seen, this simple example can become rather complex, as other statements and more bindings are added. The hierarchy can consist of several levels and is rendered complex by the fact that the bindings tie the subspace hierarchies together into a network.

While the descriptive qualities of DDS in defining the structure of a design are among its strong points, a decided disadvantage is the artificial division of a design's *behavior* into dataflow and timing, a shortcoming that both the authors (Knapp and Parker, 1984:12) and (Walker and Thomas, 1985:459) recognize. A tool such as a design simulator must recombine DDS's dataflow and timing

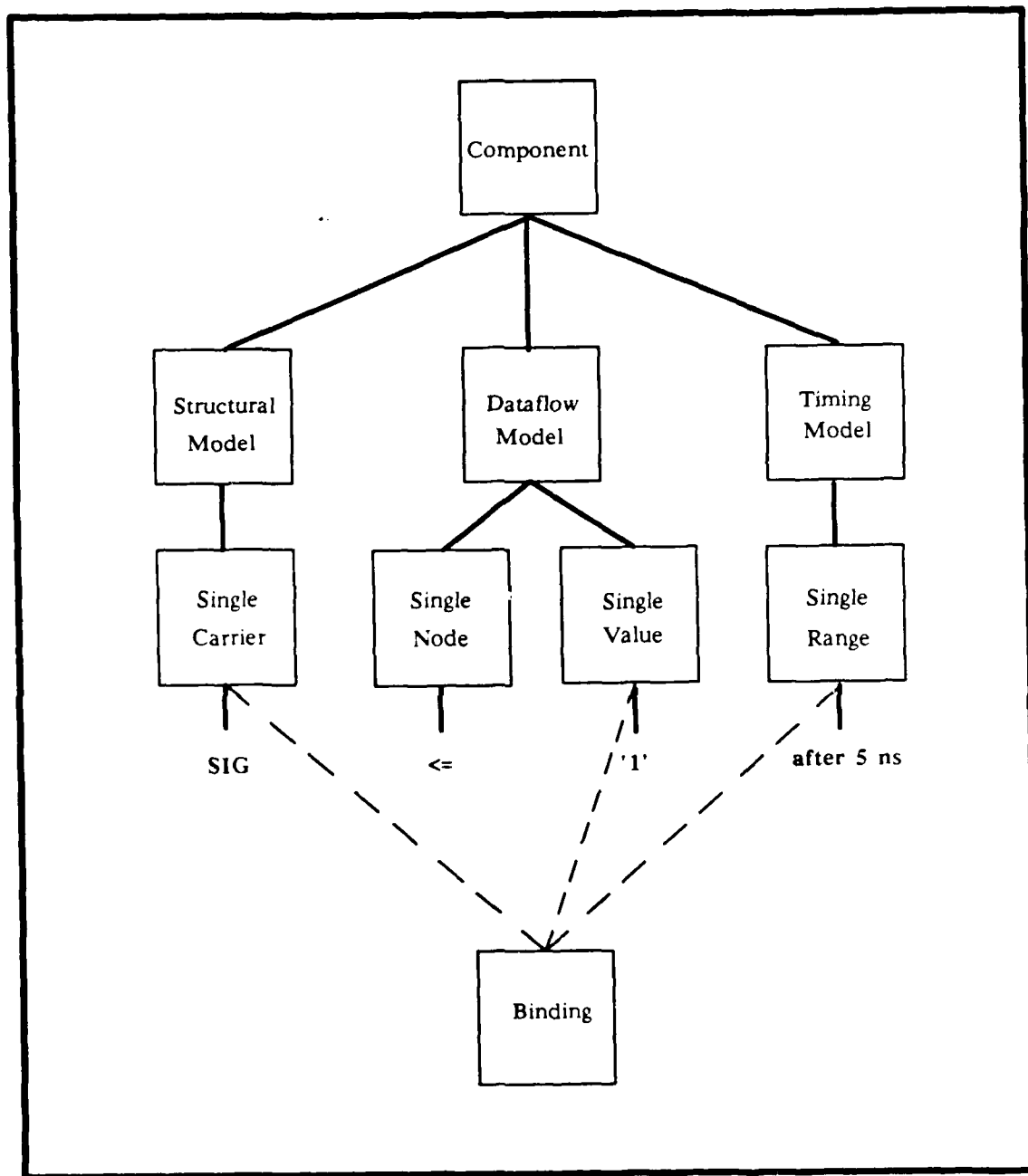


Figure 2.2 Signal Assignment Statement Modelled in DDS

information into a behavior subspace, thus making DDS harder to use. In addition, the timing subspace implementation includes "a great deal of redundancy" (Knapp and Parker, 1984:23). While the authors go on to discuss the possible tradeoffs and justify this redundancy, they also point out that it requires "a large storage space" (Knapp and Parker, 1984:23). One should also note that VHDL does not make the distinction in a design's behavior between dataflow and timing.

A further disadvantage of DDS for VHDL is the inflexibility of its control and sequencing statements, which biases DDS towards dataflow and against algorithmic representations (Knapp and Parker, 1984:6, 12): VHDL, due to its flexible descriptive characteristics, requires both kinds.

VHDL Intermediate Access (VIA). Frauenfelder found that DDS could not represent all the information specified by VHDL, specifically *dynamic sequencing* and *scheduling*. VHDL can express the next state of a simulation based on the current state; but DDS, using static sequencing, determines all future states based on initial conditions (Frauenfelder, 1986:3.15). Accordingly, Frauenfelder extended DDS to the so-called *VHDL Intermediate Access (VIA)* format (Frauenfelder, 1986:3.16).

VIA is a text file where each record (text line) can reference another record by its line number (Frauenfelder, 1986:3.16). The format for each VIA record is:

```
record-number record-type-name
( field-name-1 = field-value-1;
  field-name-2 = field-value-2;
  ...
  field-name-n = field-value-n ; )
```

Each record is numbered in the *record-number* field; the *record-type-name* determines which *field-names* will follow. Using the DDS model for hardware, a VHDL design is analyzed and broken down into three of the four DDS subspaces, dataflow, timing, and structure.² This internal DDS network is then written out in its external form, VIA. Figure 2.3 shows a simple example of a VHDL interface declaration and its DDS and VIA transformations. While this example is simple enough to readily understand, the full VIA hierarchy (shown in Figure 2.4) is many degrees more complex, both visually and conceptually. Even though VIA is an enhancement to DDS, many of the other shortcomings of DDS (when applied to representing VHDL) also apply to VIA, such as problems handling sequencing constructs and the artificial division of the design behavior. In contrast, VIA is well suited for (1) describing the structure of VHDL, (2) describing designs using dataflow implementations, and (3) providing separate information on timing requirements.

² VHDL does not currently represent the physical subspace.

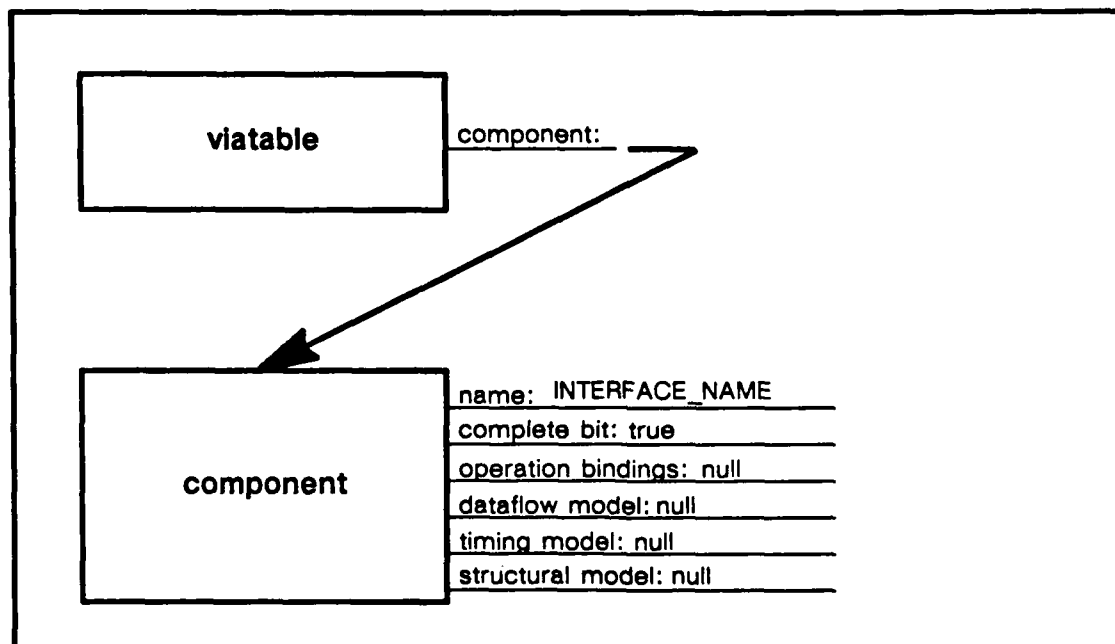


Figure 2.3 VHDL Represented in VIA/DDS (Frauenfelder, 1986:3.18)

University of Pittsburgh VHDL. Preliminary information on the University of Pittsburgh VHDL project suggests that it will be built using the source code of Frauenfelder's prototype as a basis. The proposal for this implementation describes a compiler for a subset of VHDL coupled to a mixed-mode simulator (Levitan, 1987:5). The compiler will produce "a network of primitive logic elements" that "captures the structural components of designs" and a translation of VHDL to C source code for the behavioral aspects of VHDL (sequential statements) (Levitan, 1987:3-4). After the C code is compiled, the simulator will execute the

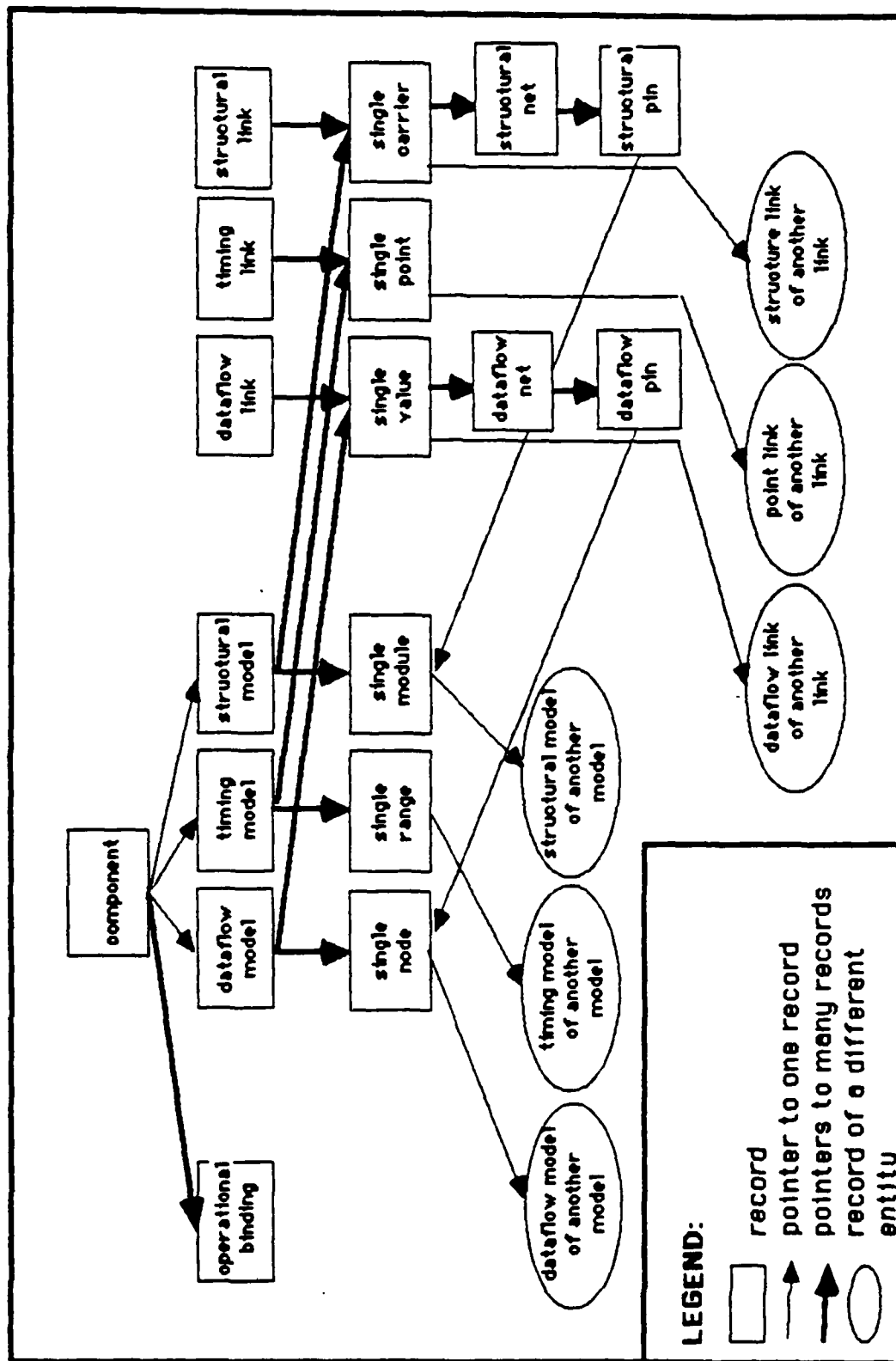


Figure 2.4 VIA Record Hierarchy; (Frauenfelder, 1986:B.3)

resulting object code whenever the corresponding behavior has to be simulated. Levitan plans to "extend the software to compile into an intermediate form suitable for a design data base" (Levitan, 1987:6).

Other HDLs. ZEUS and CONLAN are general purpose hardware description languages, similar to VHDL in their support for hardware abstraction, modular design, data abstraction and typing, functional/structural descriptions, strongly typed signals, and user-defined data types (Aylor, Waxman, and Scarrat, 1986:25-26). Both ZEUS (Lieberherr, 1985:56) and CONLAN (Piloty and Borrione, 1985:88) transform their source HDL to abstract syntax trees. ZEUS, when used for silicon compilation, stores the AST in a design database (Lieberherr, 1985:56). The CONLAN intermediate representation uses Pascal record trees and, in future implementations, will also use a design database (Piloty and Borrione, 1985:88-89).

Patois is a C based "hardware systems modeling language" designed by Dallen at Duke University. It allows a design to be modeled behaviorally, structurally, and physically from the initial specification through simulation. Its primary use is to express a behavioral description of hardware systems which can be used to drive a simulator for verification and evaluation (Dallen, 1983:4.2).

Dallen's intermediate representation is especially relevant. He uses a two-part intermediate form, representing the structure and behavior of the design. The structure portion is a list of the symbols found in the design and their definitions. Essentially, this is a listing of the analyzer's symbol table. The behavior portion is translated into *g-code*, essentially an AST, (Dallen, 1986:Section 11.1) that models the behavior of the objects in the symbol table.

Summary

This chapter has presented a survey of other VHDL research, error handling, and intermediate source forms. The work at the University of Pittsburgh on a UNIX-based VHDL analyzer/simulator was discussed. The three major types of error handling are error recovery, error repair, and error correction. This chapter also reviewed intermediate representations for HDLs, including IVAN, DDS, VIA, C source code, ASTs, ZEUS, CONLAN, and Patois. Chapter 3 continues the discussion of intermediate representations with an analysis of each representation and its suitability for use with VHDL in the AFIT VHDL Environment. I will also discuss the overall design of the AFIT VHDL Analyzer.

III. System Design

Overview of Chapter

Software engineering projects generally begin with a set of requirements defining what the end product must accomplish. These requirements and the related tasks to fulfill these requirements were presented in Chapter 1. Each task presented a problem to be solved, either with existing solutions or with new solutions. For each task from Chapter 1, I will discuss possible alternative solutions (except where one generally accepted solution exists), along with their strengths and weaknesses, and outline the proposed solution.

Find a more efficient Intermediate Representation. A major issue that had to be resolved was the Intermediate Representation which the analyzer should use. The requirements from Chapter 1 specify that it be efficient, provide easy access to the information it contains, and provide the information VHDL provides in a more efficient manner. Also, as the analyzer moves from a subset compiler to a production-quality compiler, this IR must also expand in its capabilities. The first step was to evaluate the prototype's IR, VIA, to determine whether or not to continue using it as the Analyzer IR.

The primary strength of VIA is in its structural descriptive capabilities, inherited from its DDS parentage. The fact that these capabilities were implemented first in the prototype demonstrates the facility with which VIA and DDS can structurally describe VHDL.

One shortcoming of VIA as an intermediate form in a VHDL design environment is in the area of the functional behavior of an entity. The separation of an entity's functional behavior into timing and dataflow subspaces, which are then networked together with the structure subspace, causes several problems. First, there is not a one-to-one relationship between the VHDL and the resultant VIA, thereby preventing tools that rely on this relationship (e.g., reverse analyzers) from utilizing the VIA interface (Frauenfelder, 1987). For example, a VHDL design described by VIA has many more named objects (carriers and values) than do exist in the original VHDL source. This results from the manner in which VIA handles a complex hierarchy and the bindings that exist among the levels of that hierarchy. These additional names and bindings do not map to any unique VHDL statement, but to the design as a whole. This becomes a problem when one wishes to use VIA with a reverse analyzer. A reverse analyzer, which supports the "reverse engineering" of VHSIC chips, would use the VIA as its source and output equivalent VHDL source code. Because the VIA may generate, for one VHDL statement, many records and bindings

among those records, it does not lend itself well to reverse analysis.

Second, tools that need access to both the behavior and structure of a design (e.g., simulators) have to rejoin the timing and dataflow subspaces into a behavior subspace. Refer back to Figure 2.2 in the previous chapter. In order to construct an event for simulation, the information from the timing model must be retrieved as well as the information from the dataflow model and the binding of those two models. The simulator needs the timing tied directly to transactions for efficient simulation. During simulation, as the simulation clock advances, each transaction (behavior) may be executed depending on its value (Intermetrics, 1985b:8.16-8.18). If timing and dataflow are separated, the simulator either has to search both subspaces for corresponding bindings to consolidate before the simulation begins or must perform this search for every new simulation clock cycle. This makes the design of the simulator more complex and less efficient.

The prototype implementation of VIA also proved to be rather unwieldy, though some improvements could be made. Its DDS foundation requires the use of variable length records, which, in turn, require the use of field names for each VIA record. This is because a VIA field may appear any number of times in one record, each field pointing to other

records. Thus, there is no way to know a priori how many fields a particular record may have. This requires more effort on the part of tools using VIA to parse and process each record.

Finally, the most serious problem with VIA involves its use of system resources, specifically, disk space and CPU time. The complex nature of VIA requires more computational time for its creation. A medium-sized design consisting of 1000 dataflow-type statements (signal assignment statements) took over 30 minutes to analyze on a lightly loaded VAX 11/780, with the resulting VIA output being over 3 Megabytes in size. VIA certainly cannot be lightly regarded, but, clearly, a more efficient and more general IR needed to be found.

With VIA rejected for the above reasons, there were three general categories of intermediate forms remaining for the analyzer: (1) use an attribute-annotated syntax tree similar to the IVAN format, (2) use a high order language similar to that being used at the University of Pittsburgh, or (3) design a new intermediate form, possibly using abstract syntax trees.

Recall from Chapter 2 that IVAN is an annotated Ada-based intermediate form. Tools requiring access to the original VHDL syntax, like reverse analyzers and VHDL optimizers, can use the IVAN format. Another strength (and

weakness) is that it is Ada-based, i.e., that it uses an Ada compiler. This dependency insures IVAN portability among systems with validated Ada compilers, but not all UNIX installations have such compilers. Since this project must use *common* UNIX tools, IVAN is an unlikely choice. Finally, as mentioned in Chapter 2, IVAN requires an ancillary design library and library manager. Because the AVE IR itself is to serve as the design library, IVAN could not be used in the AVE without extensive modification or the development of supplementary tools.

Plans for the University of Pittsburgh VHDL analyzer call for separating the VHDL into a network of structural descriptions (which Levitan calls "primitive logic elements") and C source code (to represent the behaviors) (Levitan, 1987:3). I will refer to this intermediate form as VHDL/C. Independent of his effort, I developed a similar intermediate form using C, with the following conclusions:

1. The use of C is in keeping with the philosophy of the UNIX environment--every UNIX operating system, being based on C, has a C compiler. This insures portability and avoids one of the limitations of IVAN, which uses Ada.

2. The division of VHDL, as by Levitan, into only structure and behavior components, is a sound decision, supported in the Language Reference Manual. The implementation of the simulator is thereby simplified in

that whenever a design entity's behavior has to be simulated, the compiled C code can be run instead. This allows an analyzer/simulator system to be more quickly designed and implemented.

3. A major weakness of VHDL/C is that several VHDL constructs have no direct C counterparts. These include signal attributes, physical types, array slices, and enumeration types (in some C implementations). While block statements can be nested in VHDL, C functions cannot. Levitan plans to avoid this issue temporarily by supporting "only simple typing, scoping and visibility" and add the full language at a later time (Levitan, 1987:5).

4. Another major weakness results from the first: since this intermediate form is partly a C translation of the VHDL source, tools relying on a representation of the original VHDL source cannot use VHDL/C. Examples of such tools are reverse analyzers and VHDL optimizers. Because of the nesting problem mentioned in the preceding paragraph, the normal hierarchy of design functions has to be "flattened" to be represented in C. This, for practical purposes, irrecoverably masks the structure, though not the behavior, of the original VHDL.

5. Finally, the compiled C code hides much of VHDL's behavioral information on design entities. While simulation tools do not need this information, tools such as Decker's

microcode retargeter cannot use VHDL/C. This is true because the microcode retargeter does not simulate a VHDL design but relies on VHDL to provide a description of a chip's structure and micro operations, which is then used to retarget microcode for that chip (Decker, 1986:9-12). Therefore, behavioral information, as well as a structural description, must be readily available in a general and efficient intermediate form. To use VHDL/C in this manner would require another analysis, this time from the C source code to some other form. This is not cost effective if other alternatives can be found.

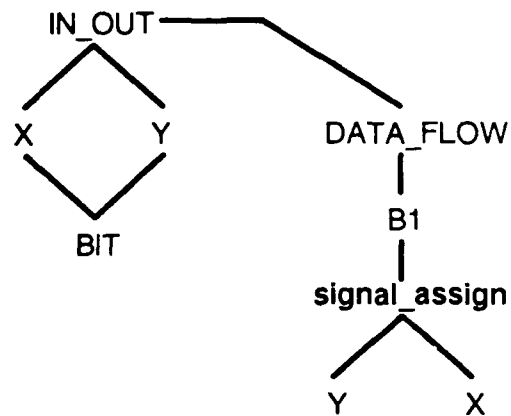
In summary, based on the above discussion, VHDL/C, while suited to a "'stand alone' compiler/simulator system" (Levitan, 1987:5), cannot serve as the interface for a more general design environment, such as the AVE.

ZEUS, CONLAN, and IVAN (to a degree) use abstract syntax trees (ASTs) as their Intermediate Representations. The main strength of this approach is that algorithms exist for creating and maintaining ASTs (e.g., Noonan, 1985), as do ways for storing and retrieving ASTs to/from disk files. An AST, being created directly from the VHDL syntax, can provide VHDL's structure to those tools that need to see this information. Figure 3.1 shows an example of a VHDL design and the resulting AST representation, clearly demonstrating the correlation between the two. On the other

VHDL Simple Design:

```
entity IN_OUT
  ( X : in BIT;
    Y : out BIT)
is
end IN_OUT;
architecture DATA_FLOW of IN_OUT is
  B1: block
  begin
    Y <= X;
  end block B1;
end DATA_FLOW;
```

Abstract Syntax Tree:



C Representation:

```
IN_OUT (X, Y)
  int X, *Y;
  { *Y = X; }
```

Figure 3.1 VHDL Source and Resulting AST and C Code

hand, an AST is not as concise a representation of VHDL behavior as C source code (which is also shown for comparison). But an AST can provide more explicit information, such as resultant types of expressions and operations, which can be attached as attributes to each tree node.

The biggest problem with ASTs is that, while VIA and DDS separate VHDL designs into too many subspaces, an AST performs no separation at all. This lack of separation, in effect, hides part of the information that VHDL supplies, the division into structure and behavior; a requirement of the IR (from Chapter 1) is that it supply all the structure and behavior information contained in the original VHDL. Therefore, an IR with the simplicity of an AST coupled with separate structure and behavior subspaces is needed.

This requirement can be met using the Intermediate Representation of Patois, which provides several advantages. First, it reduces the complexity of the analyzer by limiting the number of subspaces to manage to only two. Second, it offers a more universal solution to the interface problem: VIA tried to fit VHDL into a *model* which was not well suited to the needs of other AVE tools and VHDL/C tried to involve the analyzer in areas best left to the simulator (*i.e.* generation of C code, which is still the best way to handle the simulation). Also, Patois is already UNIX-based (being

written in C), virtually eliminating portability problems. A new Intermediate Representation, based on Patois, was developed and was also called VIA. This new VIA retains the overall structure of Patois, having structure and behavior divisions, but was modified specifically for use with the VHDL language. First, the structure division was modified in order to accommodate the types of objects found in VHDL, which are different from those found in Patois. This structure division is called the *symbol table* (SYMTAB). Second, new record types were added to the behavior division, since VHDL is a more complex language in this respect than Patois. The behavior division is called the *operation table* (OPTAB).

Document the IR for use by other tools in the AVE. The definition for VIA can be found in Appendix B.

Adapt the prototype for use of the new IR. After having decided upon using the Patois-based IR as the new VIA, the prototype analyzer was modified to generate the new IR. Due to the prototype's modular design, this involved only replacing the modules that generated VIA with ones to generate the AST and modifying the semantic checking routines. The resulting design is shown in Figure 3.2.

The lexical analyzer produced by the UNIX utility *lex* (Lesk and Schmidt, 1978) and the parser generated by *yacc* (Johnson, 1978) form the foundation of the analyzer. These

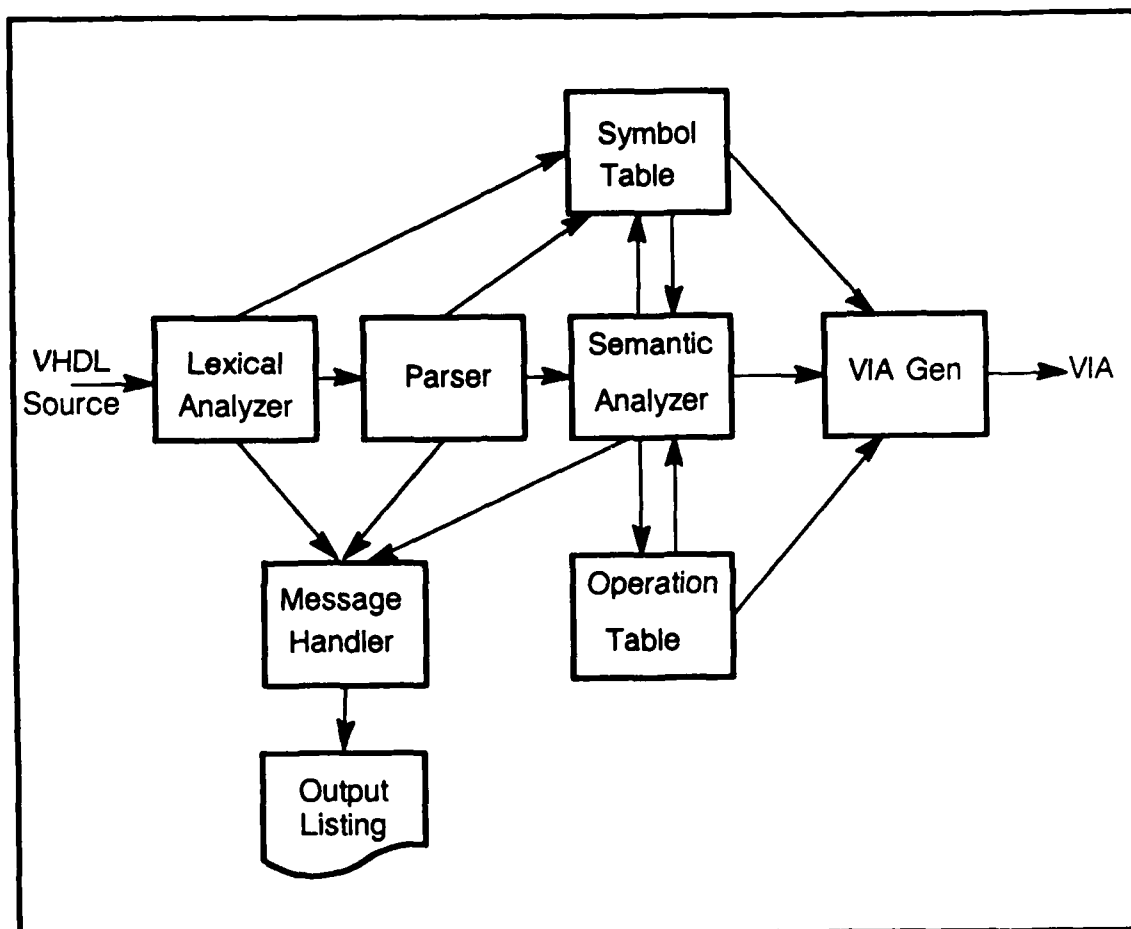


Figure 3.2 Analyzer Design

tools are common UNIX utilities and the *de facto* standards for compiler development. The *lex* scanner groups the individual characters of the VHDL source into lexical elements (keywords, identifiers, literals, etc.) called *tokens*. The *yacc* parser checks the syntax or the ordering of the tokens in a VHDL design and creates a stack structure of the results of the analysis of each VHDL grammar rule. As each statement of the VHDL source is scanned and parsed,

yacc passes this parse stack to the semantic analysis portion of the analyzer, which determines the semantic validity of the statement. Symbol table and operation table routines handle the creation of their respective tables as each VHDL statement is analyzed. Errors, warnings, and other messages are sent through a separate message handler, to be output with the output listing. When a design has been analyzed with no errors, the code generator routines read the SYMTAB and OPTAB to generate the VIA.

The original structure of the symbol table was also improved for efficiency. Execution profiling showed that the prototype analyzer spent much of its time either entering data into or retrieving data out of the symbol table. The prototype design was based on a kernel for a C compiler, using its simple linked-list symbol table, modified for use with VHDL (Frauenfelder, 1986:3.4, 3.10). A more efficient approach is to use a hash table for identifier lookup and a display table to implement scope and visibility. The hash table allows (in most cases) direct access to an identifier, therefore, yielding constant $O(1)$ lookup times versus $O((n + 1)/2)$ time for a sequential search of table of size n (Tremblay and Sorenson, 1985:429, 450). The display table is a stack of contexts. As declarative regions are entered and exited, these contexts are pushed and popped, causing the declarations they contain to become visible and hidden. Routines to access a symbol

table implemented in this manner are simpler to construct and more efficient than ones that have to work with a linearly-ordered symbol table (Aho, Sethi, and Ullman, 1986:429).

Add error handling capabilities to the analyzer. The alternative solutions for error handling discussed in Chapter 2 are error recovery and error repair. In the academic environment, students learning to program (whether in a software or a hardware programming language) are more prone to mistakes than in a commercial environment where the programmers are expected to have more expertise. Error repair is probably not cost effective because any error handling technique that flags the errors (for the students to later correct) will suffice in a learning environment (Aho, Sethi, and Ullman, 1986:164). Therefore, only error recovery techniques were considered.

The two main error recovery techniques discussed in Chapter 2 were panic mode and the use of error tokens. As noted there, a program that halts after detecting the first error is not acceptable. Since error recovery should try to uncover as many errors as possible, the panic mode method of error recovery is unsuitable. Because the use of error tokens is "built-in" to the UNIX tool used to create the VHDL parser (yacc) (Schreiner and Friedman, 1985:Ch 4), this means of error handling was selected.

Construct/obtain a test suite for the analyzer which tests for syntax and static semantics conformance to the VHDL Version 7.2. The actual test suite for the VMS VHDL analyzer was used to test the analyzer for conformance. As was mentioned in Chapter 1, VHDL designs by Dukes, augmented by designs from other VHDL graduate students, formed the test suite for performance and integration testing. More on testing can be found in Chapter 5.

Add full language capabilities to the analyzer. After the design for the VIA interface was selected, adding new language constructs to the analyzer became more of an implementation problem than one of design. The methods proposed by Frauenfelder which were outlined in Chapter 1 were used in the implementation and are detailed in Chapter 4.

Perform analyzer and system integration tests. The conformance tests, mentioned above, verify only that the analyzer can process correct VHDL and detect incorrect VHDL. Other tests and evaluations must be performed before the analyzer can be deemed "production-quality." These tests evaluate how well the analyzer performs, how well it conserves system resources, how portable it is, and how well it performs with other AVE tools. The results of these tests, as well as the analyzer conformance tests, are found in Chapter 5.

Document the Results. This chapter, and Chapters 4 and 5, document the major design decisions and test results of this project.

Summary

This chapter has presented the overall design of the AFIT VHDL Analyzer with emphasis on the manner in which alternative solutions were evaluated and a new IR was selected. The selected intermediate form is based on the intermediate form used in Dallen's Patois. The details of the analyzer's implementation of the new IR, also called VIA, are presented in the next chapter.

IV. Detailed Design

Overview

In Chapter 1, requirements the Analyzer had to meet in order to be considered production-quality were discussed. To satisfy these requirements, certain tasks (extending the VIA, adding error recovery, completing the language subsets, and testing) had to be completed; and for each task, design solutions were selected. This chapter will discuss those design solutions in the implementation of VIA and the remaining language subsets.

VIA Modification

After the form for the intermediate representation was selected, its specification was documented (and can be found in Appendix B). It was then determined that three steps would accomplish the modification of VIA/DDS. They were:

1. Determine the information provided by VHDL and whether this information is structural, behavioral, or a binding of structures. Bindings connect structures together through the VIA AST.
2. Map the VHDL structure to the VIA symbol table, VHDL behavior to the VIA AST, and bindings of VHDL structures to the AST which connects entries in the symbol table.
3. Code and test the modules to generate VIA.

This was an iterative process for each of the language subsets to be implemented. The following sections discuss these steps in more detail.

1. Determine the information provided by VHDL and whether the nature of this information is structure, behavior, or a binding of structures. VHDL can be mapped into two subspaces--structure and behavior^{1,2}. For example, the interface declaration from the full-adder design (Figure 4.1) is mainly structural. It specifies the name of the interface and the names and types of its ports. On the other hand, the signal assignment statement in the architecture description *DATA_FLOW_IMPL* shows behavior--the value of the expression on the right hand side of the statement is assigned to the signal variable on the left hand side. For each VHDL construct, a similar analysis, using the information provided in the *VHDL Language Reference Manual* and *User's Reference Guide* (Intermetrics, 1985b and 1985d), was performed, and the information that VIA needed to provide was added to that construct's VIA definition. This was generally a straightforward process.

¹ Both the Walker/Thomas model (mentioned in Chapter 2) and DDS include a *physical* subspace. At this time, VHDL does not describe the physical aspects of a design, such as the wiring geometry (Nash and Saunders, 1986:65). Until this capability is added to VHDL, the physical subspace must be ignored.

² Nash and Saunders provide this mapping (including the physical domain) for each level of abstraction from architecture down to the circuit level (Nash and Saunders, 1986:55). In the case of component instantiations or procedure/function calls with parameters, each language construct provides information that binds two structures together.

VHDL Full Adder:

```
entity FULL_ADDER
  ( X, Y : in BIT;  -- one-bit addends
    CIN : in BIT;  -- carry in
    SUM : out BIT;  -- one-bit sum
    COUT: out BIT)  -- carry out
  is end FULL_ADDER;
architecture DATA_FLOW_IMPL of FULL_ADDER is
  BLOCK_1: block
    signal C: BIT;  -- Local signal declaration
  begin
    SUM <= X xor Y xor CIN after 5 ns;
    C   <= ( Y and CIN ) or ( X and CIN ) or ( X and Y );
    COUT <= C after 6 ns;
  end block BLOCK_1;
end DATA_FLOW_IMPL;
```

Figure 4.1 Full-Adder Example (Intermetrics, 1985c:1.3-1.5)

due to the aforementioned mapping. One problem encountered was the combination of *directives* and *specifications* with *declarations* in several places in the VHDL grammar. While declarations defined entries in the symbol table, directives and specifications define operations and, therefore, are mapped into the OPTAB AST. This caused a problem in that the routines processing declarations return pointers to the symbol entry while the routines processing AST entries return array indices (integers). By requiring directives and specifications to follow all declarations, this problem was solved.

2. Map the VHDL structure to the VIA symbol table and VHDL behavior to the VIA AST. Using the information from Step 1, each VHDL construct was mapped into VIA. The structural portions of each construct would be entered into the symbol table to be written to the VIA SYMTAB upon analysis completion, while the behavioral portions generated nodes of the abstract syntax tree (OPTAB). Bindings became nodes of the AST with leaves that referenced the symbol table portion of VIA. An example of a binding is a *function call* with a parameter list. Each *formal parameter* is bound to an *actual parameter*, but this binding is created in the operation table because the entire *function call* statement is an operation.

3. Code and Test Generate VIA Modules. Finally, test cases were written for each construct added, the Analyzer processed each test case, and the results were checked and verified. Chapter 5 describes the testing procedures and sample test cases.

Use of lex and yacc

Before discussing the implementation details in which VHDL is mapped by the Analyzer to VIA, a short discussion of the use of *lex* and *yacc* will make that section more understandable. The UNIX utility *lex* (Lesk and Schmidt, 1978), when given an input file of *regular expressions* that maps character sequences into *identifiers* and *literals* (called *tokens*), generates a C routine (*yylex*) that will perform

that mapping³. The UNIX utility *yacc* (Johnson, 1978) uses an input *grammar* to construct a C routine (*yyparse*). The grammar *yacc* uses is similar to Backus-Naur Form (BNF), which is used to define many current programming languages (VHDL among them). Thus, given a BNF for a language, it is a straightforward task to construct a *yacc* input for that language⁴. The *yacc* grammar is composed of one or more *rules* or *productions*. These rules specify how a particular language statement (or *construct*) is put together. For example, a simple English sentence could be represented by:

<sentence> ::= <noun> <verb> <object> '.'

This shows that a *sentence* is a *noun* followed by a *verb*, an *object*, and end with a period ('.'). The elements *sentence*, *noun*, *verb*, and *object* are called *nonterminals*, which means they are, themselves, composed of other elements. The period is a *terminal* symbol--it cannot be decomposed. When *yacc* parses a *noun* followed by a *verb* followed by an *object* followed by a period, it *reduces* this rule to a *sentence*. In the example grammar, *sentences* may be reduced to *paragraphs* and *paragraphs* to *chapters*. In this manner, an entire "program" may be reduced to one symbol, known as the

³ Regular expressions are shorthand for specifying different sequences of characters. Identifiers are names given to designs, packages, objects, types, etc. Literals are numbers or characters that represent themselves, such as 0, 3.141579, 'A', and "this character string".

⁴ *yacc* produces a parser for LALR(1) languages. The differences between LR, LL, and LALR languages are outside the scope of this discussion and should not hinder its understanding.

start symbol. Of course, if the input being parsed fails to match any rule in the grammar (if, for example, an *object* precedes the *noun*), an error occurs and *yacc* displays a user generated message (Aho, Sethi, and Ullman, 1986:264-266).

yacc allows *semantic actions* to be embedded in the grammar. Using the above example, the *yacc* input would be:

```
sentence :  
    noun  
    verb  
    object  
    '.'  
        { $$ = check_verb($2); }  
    ;
```

As *yacc* parses the input, it creates a *stack* of the results of that parse. This stack is available for use as input to C routines, such as "check_verb" in the above example. Here, the value of *verb* (denoted by "\$2") is passed to "check_verb" (which could check to see if the verb takes an object or not). The return value of "check_verb" becomes the new stack value ("\$\$") (Aho, Sethi, and Ullman, 1986:260). For a more complete discussion of the use of *yacc* and *lex* with compilers, see (Schreiner and Friedman, 1985) or (Aho, Sethi, and Ullman, 1986:257-266).

Language Implementation

The implementation of the remaining subsets of the VHDL required the following steps for each construct in each subset:

1. Check that the syntax is correctly defined for yacc in the prototype⁵. If not, correct it, using the VHDL grammar given in the Language Reference Manual (Intermetrics, 1985b:Appendix C).
2. Determine the resulting VIA (from the above-described steps to modify the VIA) for each VHDL construct. Design, code, and test modules to generate that VIA.
3. Determine what values would be left by yacc on its value stack upon completion of the analysis of each construct. These values are the results (i.e., symbol table or tree table references) from earlier grammar reductions.
4. Determine what *semantic actions* need to be taken during the parse of this construct. For example, in assignment statements, the type of the right hand side must match the type of the left hand side or a *semantic error* occurs.
5. Design, code, and test each function that performs a semantic action and insert it into the yacc grammar so that the parser performs this action at the appropriate time in the analysis.
6. Add error recovery for each construct.
7. Design test cases for each construct, run the Analyzer against each test case, and verify the results.
8. After each construct is successfully tested, inform the simulator design team so they could also test the new construct with the simulator (Kodama, 1987).

This stepwise approach insured that problems and errors were detected and corrected before they had a chance to impede the progress of the project. It also insured that each construct was implemented completely before attacking the next. It involved the design team of the AFIT VHDL simulator so that as each VHDL construct was added, the

⁵ The prototype analyzer was designed to accept the entire VHDL. The yacc input was double-checked to insure that there were no errors.

Analyzer produced the correct VIA and the simulator could be upgraded to process the Analyzer's output.

Implementation Examples

Behavior Example: An example of the way a *if_statement* was implemented will help demonstrate the implementation process for behavior language constructs. The BNF definition of the *if_statement* is:

```
if_statement ::=
    if condition then
        sequence_of_statements
    { elsif condition then
        sequence_of_statements }
    [ else
        sequence_of_statements ]
    end if ;
```

1. Check that the syntax was correctly defined for yacc. From the above BNF grammar, a corresponding yacc grammar was written. The yacc input for an *if_statement* is given as

```
if_statement :
    IF
    condition
    THEN
    ..ELSIF__THEN__seq_of_stmts..
    .ELSE__seq_of_stmts.
    END
    IF
    Semicolon
    {
        $$=if_statement($2, $4, $5, $6);
    }
    ;
```

Upper case names (such as IF and THEN) are tokens passed from lex; lower and mixed case names (condition, Semicolon,

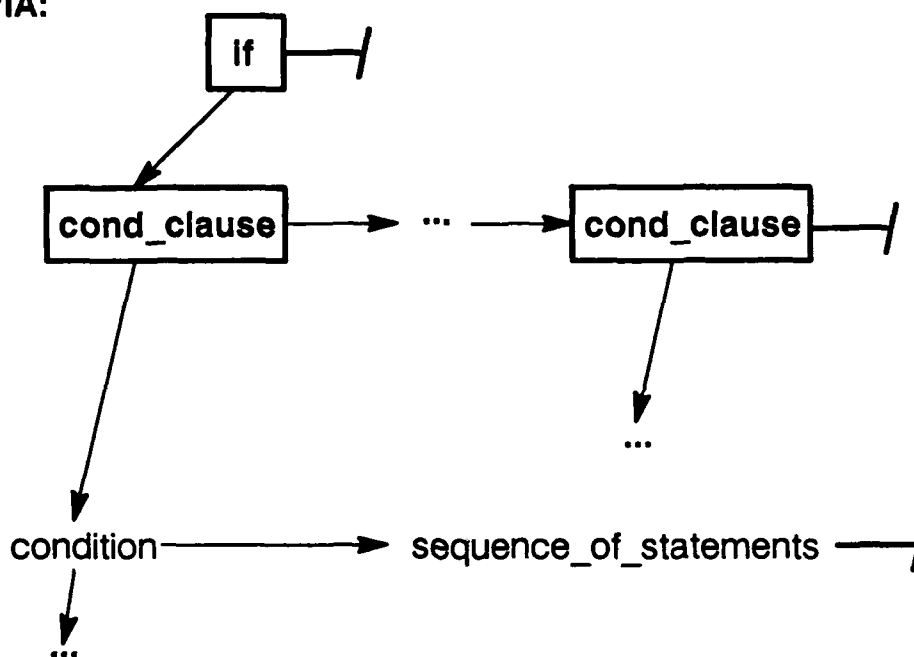
and `.ELSE__seq_of_stmts.`⁶) denote *productions*; and the statements between the braces (`$$=if_statement($2,...,$6);`) are C functions (defined in another part of the Analyzer) that will be performed when the parser accepts the immediately preceding production. The "\$\$" represents the value of the yacc stack for this construct and, in this example, is a pointer to the AST generated and returned from the C function "if_statement." For the *if_statement*, these C functions originally (in the prototype) displayed only a message "If statement not implemented" and returned a null value. Comparisons with the VHDL LRM grammar and testing showed that the yacc grammar was correct.

2. Determine the resulting VIA for each VHDL construct. Design, code, and test modules to generate that VIA. The VIA that should be generated for the *if_statement* is given in Figure 4.2. This was designed using generally accepted methods of generating ASTs from languages like Pascal and Ada (which VHDL closely resembles) (Aho, Sethi, and Ullman, 1986:287-290). The yacc rules for *condition* and *sequence_of_statements* (which were designed in the same manner as the *if_statement*) will generate the appropriate

⁶ yacc allows non-terminals to be named using alphanumerics, the underscore, and the period. A naming convention was used whereby non-terminals that could be repeated zero or one time began and ended with a single period (representing the beginning and ending brackets in Backus-Naur Form); a non-terminal that could be repeated zero or more times began and ended with two periods (for enclosing braces); and a non-terminal that defined a list that began with a comma and could be repeated any number of times, began with three periods (signifying an opening brace and the comma) and ended with two periods (the closing brace). In this manner, the action of every yacc grammar rule could be determined from its name.

VHDL Source: `if condition then`
 `sequence_of_statements`
 `elsif condition then`
 `sequence_of_statements`
 `else`
 `sequence_of_statements`
 `end if;`

VIA:



Lower case letters denote the root of a subtree of one or more nodes.

`...` Denotes zero or more nodes.

`—|` Denotes a NULL pointer.

Figure 4.2 VIA for `if_statement`

VIA when executed and place pointers to that VIA on the *yacc* parse stack.

3. Determine what values would be left by yacc on its value stack upon completion of the parse of each construct.

The value stack returned by *yacc* for the *if_statement* is:

- (1) the token IF
- (2) the pointer to the AST node for *condition*
- (3) the token THEN
- (4) the pointer to the AST node for the first statement in *sequence_of_statements*
- (5) the pointer to the AST node for a condition node for the ELSIF part
- (6) the pointer to the AST node for the first statement in the ELSE part
- (7) the token END
- (8) the token IF
- (9) the token Semicolon

The types of values returned by nonterminals are determined by the return types of the corresponding semantic action routines written by the user. The values for the terminals (tokens) are integers determined by *yacc* from a list of terminal symbols supplied by the user.

4. Determine what semantic actions must be taken during the analysis of each construct. In this construct, only the *condition* must be checked for being a *boolean* type (it has only the values *TRUE* or *FALSE*) and the grammar rule that parses *condition* handles that chore. Therefore, no further semantic actions need be taken for this example. The parser must connect the AST nodes returned on the parser stack (from the analysis of *condition* and any ELSIF and/or

ELSE statements) in the proper manner to construct the VIA tree given in Figure 4.2.

5. Design, code, and test each function that performs a semantic action and insert it into the yacc grammar so that the parser performs this action at the appropriate time in the analysis. In this case, the routines to create and connect AST nodes were already coded and tested. Then, before error recovery was added, a few sample test cases (with no syntax errors) were run to insure that the `if_statement` was performing correctly.

6. Add error recovery for this construct. As was mentioned in the previous chapter, yacc supports syntactical error recovery through the use of a special token called `error`. In practice, error recovery consisted of adding another sequence to the end of the above described `if_statement` syntax. The complete yacc grammar for the `if_statement` is now:

```
if_statement :
    IF
    condition
    THEN
    ..ELSIF__THEN__seq_of_stmts..
    .ELSE__seq_of_stmts.
    END
    IF
    Semicolon
    {
        $$ = if_statement($2, $4, $5, $6);
    }
| IF
  error
  Semicolon
  ;
```

This addition to the grammar allows the parser to accept either the correct *if_statement* or the reserved word "if" followed by any syntax error (denoted by the token "error"⁷) and continue parsing until it detects a semicolon.

Finally, Steps 7 and 8 tested the new construct (against correct and incorrect test cases), and when it passed, the simulator design team was notified of an update to the Analyzer.

Structure Example: In a similar manner, each language construct that described a structure in VHDL was implemented. Recall in Chapter 2, an example of DDS was presented using a VHDL *signal* and *signal_assignment statement*. A *signal_declaration*, then, defines that type of VHDL object and creates a symbol table entry for a list of one or more names (*identifiers*) which will be declared as signals. The BNF notation (step 1) for this grammar rule is:

```
signal_declaration ::=  
    signal identifier_list : subtype_indication ;
```

and the yacc grammar is:

⁷ The token *error* generates an error message that lists the tokens the parser was expecting.

```

signal_declaration :
    SIGNAL
    identifier_list
    Colon
    subtype_indication
    Semicolon
    {
        $$ = signal_declaration($2, $4);
    }
;

```

The VIA for this construct (step 2) consists of one or more symbol table entries, each containing the name of the signal and a reference (pointer) to the type given by *subtype_indication* (Figure 4.3). *yacc* returns the following value stack (step 3):

- (1) the token SIGNAL
- (2) a pointer to a linked list of identifiers
- (3) the token Colon
- (4) a pointer to the symbol table entry for *subtype_indication*
- (5) the token Semicolon

There are several semantic actions (step 4) that must be performed. First, locations for each identifier in *identifier_list* must be created in the symbol table. Since we are declaring new object names, no other definitions using the same name in the same design should exist; if this is not true, an error message (generated by the symbol entry create routine) indicates that the object is declared twice. VIA categorizes each structure in VHDL into *classes* and *kinds*. A class may be an *object*, (design) *unit*, or *type*. A kind may be an object like a *variable*, *signal*, or *constant*; a unit like an *interface*, *architecture*, or *package*; or a

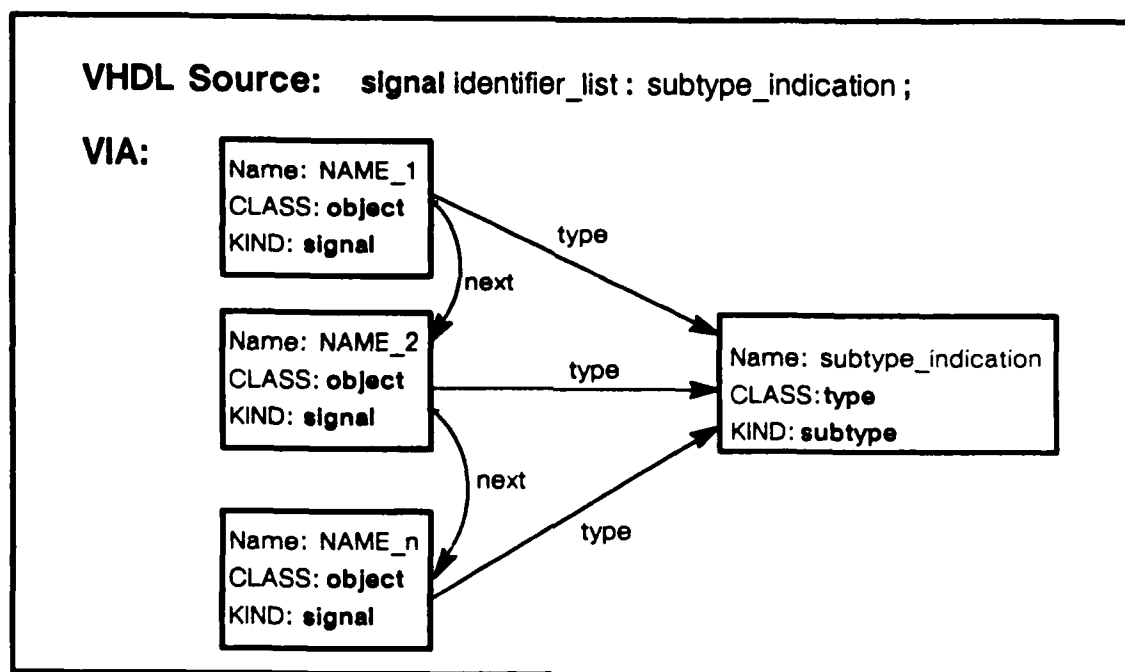


Figure 4.3 VIA for *signal_declaration*

type such as an *integer range*, *enumeration*, or *array*. In the above example, for each identifier, a symbol table entry is created for an *object* class of kind *signal*. The type of each identifier is referenced by *subtype_indication*. If this reference is NULL, *subtype_indication* was not defined earlier (and has already been noted by an error message). When this error occurs, the type of each identifier is a reference to ANY_TYPE. This special entry allows the identifiers to be used later in the analysis as if they had been properly declared, without causing a rash of error messages.

Next, since the lookup and creation of symbols are predefined symbol table primitives, no other semantic routines needed to be implemented (step 5). Error recovery (step 6) was added using a grammar rule similar to the *if_statement*. Test cases were created, the proper operation of this construct was verified (step 7), and the definition of this construct was given to the simulator design team (step 8). In a similar manner, using the information supplied in the VHDL LRM, the rest of the VHDL language subset was implemented and tested.

Summary

This chapter discussed the methods employed in the implementation of the design decisions made in Chapter 3. The steps taken to create the VIA for new VHDL constructs and to implement new language subsets to the Analyzer were outlined and examples of behavioral and structural constructs were shown. Chapter 5, Test and Analysis, will discuss the ways the Analyzer was tested and the manner in which it met the production-quality requirements.

V. Testing and Analysis

Introduction

This chapter will describe the testing performed on the UNIX VHDL Analyzer and the analysis of those test results. It will show what tests were needed to demonstrate that the thesis requirements were met. It will also show how those tests reflected the quality of the thesis product, based on the criteria established in Chapter 1.

Thesis requirements

In Chapter 1, the requirements for this thesis were developed. This section reviews the thesis requirements and shows how each one was tested for completion. These requirements are outlined below:

Generalization:

- (1) Run under UNIX.
- (2) Run on several different computer architectures.
- (3) Analyze both correct and incorrect VHDL.

Testing:

- (4) Pass VHDL Test Suite.

Documentation:

- (5) Be well documented.

Maintenance:

- (6) Include inline comments and module specifications.

Interfaces:

- (7) Produce Intermediate Representation providing analyzed VHDL information to other environment tools.
- (8) Produce an efficient and easy to use IR.

System Resources:

- (9) Process VHDL designs within a "reasonable" time period.
- (10) Conserve the use of main memory and secondary storage.

System Integration:

- (11) Must show "end-to-end" use of the AFIT VHDL Environment, specifically, the simulation of VHDL designs.

From this list, we see that the requirements for Documentation (5) and Maintenance (6) only require the submission of material. Interface requirement (7) requires that an IR be produced; its quality is determined in requirement (8). Testing ensured the remainder of the requirements were satisfied. These tests were divided into four categories based on the requirement areas: conformance tests, portability tests, performance tests, and integration tests.

Conformance tests. (Requirements 3 and 4). Conformance tests ensured that the Analyzer conformed to the specification of VHDL given in the VHDL Language Reference

Manual (Intermetrics, 1985b) and met the production-quality requirements (3 and 4) listed above. As each language construct was added to the Analyzer, these tests were performed to verify that no errors were introduced during coding and that the new code did not adversely impact existing code in terms of efficiency and module interfaces (parameter passing). For example, early releases of the Analyzer only recognized *simple names*. Later releases of the Analyzer were able to handle *indexed names* (array names), which are represented in VIA differently from simple names. Each routine that processed any VIA "name" node was modified to now also handle array names. Testing ensured that all the necessary changes had been made. It was only when tests showed the Analyzer conformed to the LRM that other tests could be performed. The final phases of conformance testing used the VMS Analyzer test suite described in (Intermetrics, 1984b).

Portability tests. (Requirements 1 and 2). These tests ensured that the Analyzer performs under UNIX bsd version 4.2 (Bell, 1983) and on various computer configurations. Passing these tests showed that the Analyzer design and implementation could be moved to other UNIX sites, thus meeting the generalization criteria for a production-quality tool.

Performance tests. (Requirements 8, 9, and 10). These tests ensured that the Analyzer's performance met the

project's requirements for processing speed, memory usage, and disk usage.

Integration tests. (Requirement 11). The final set of tests involved the AFIT UNIX VHDL simulator (Kodama, 1987). These tests were "end-to-end" simulations of several VHDL designs with the results manually verified and compared to simulations of the same designs run under VHDL/VMS. These tests confirmed the interoperability of the system interface (VIA) with the simulator.

Conformance Testing

Introduction. Conformance testing showed the degree to which the Analyzer met the VHDL LRM 7.2 "standard". This testing was similar to the testing in the Ada Compiler Validation Capability (ACVC) (Goodenough, 1986), which is itself based on the Pascal Validation Test Suite (Wichmann and Ciechanowicz, 1983). Both ACVC and the Pascal Test suite involve three phases: LRM conformance, error handling, and implementation restrictions¹.

LRM Conformance. Testing for conformance to the VHDL LRM involved: selecting a part of the language to be tested, determining what objectives must be met in order that the selected part is fully tested, and constructing

¹ Because this last phase required executable code (such as that produced by an Ada or Pascal compiler) and the Analyzer only produces a non-executable intermediate form, this test was not performed on the Analyzer.

tests for those objectives. This is the manner in which the test suite for the VMS VHDL system was constructed (Intermetrics, 1984b). The VMS Analyzer test suite contains two types of tests: *short* and *error*. The short tests contain correct VHDL and must generate the correct output VIA. The Analyzer must not detect any syntactic or semantic errors for any fully implemented construct. Since this implementation is a subset VHDL analyzer, semantic errors were allowed for non-implemented features of the language, but no syntactic errors. The error tests contained syntax and semantic errors which are supposed to be detected by the Analyzer. For those features of the language that were fully implemented, all syntactic and semantic errors had to be detected. For partially and non-implemented language constructs, an error test that caused no error messages was allowed to pass, but an error test that caused an abnormal termination of the Analyzer failed. For example, a simple objective is:

Show that the Analyzer will accept the syntax for the simplest interface declaration, which is

```
entity identifier is end ;
```

This objective can be tested by constructing a short test using the above interface declaration (substituting a valid name for *identifier*) and submitting it to the Analyzer (the actual test run is shown in Listing 5.1). If the correct

Listing 5.1 Syntactic Conformance Testing

```
verbose on
AFIT VHDL Analyzer Revision: 3.0

[1] --TEST 1.1.1-1, CLASS=CONFORMANCE, SECT=Interface Declaration
[2]
[3] --: This program tests the minimal interface declaration.
[4]
[5] entity i1 is
[6] end ;
Number of errors detected: 0
Generating VIA file...done
VHDL Analysis complete.
```

VIA representing the entity interface is produced, the Analyzer passes. Any other result indicates failure².

For a semantic error test, an objective might be:

Ensure that, in an architecture body declaration, the entity name of the interface is visible and has been analyzed before the architecture.

The test for this objective is shown in Listing 5.2. For the VMS Analyzer test suite, each objective and corresponding test for the rest of the language was determined from the definition of VHDL in the Language Reference Manual.

In using the VMS Analyzer test suite, several modifications had to be made. First, the tests that only

² This is known as *black-box* testing. It assumes the internal structure of the program is hidden and only tests for the results of executing that program.

Listing 5.2 Semantic Error Testing

verbose on

AFIT VHDL Analyzer Revision: 3.0

```
[1] --TEST 1.2.1-3, CLASS=ERROR, SECT=Architectural Body Declaration
[2]
[3] --: This program tests that the correct entity name is found for a
[4] --: particular architectural body declaration.
[5]
[6] -- This program requires also the minimal block and process statements.
[7]
[8] -- When separate compilation is completed, the interface declaration
[9] -- preceding the arch declaration can be deleted.
[10]
[11] entity i1 is
[12] end i1;
[13]
[14] architecture t1p2p1d3 of i2 is
[15]   block1:
[16]     block
[17]     begin
[18]       process
[19]       begin
[20]         null;
[21]       end process;
-->[warning] line 21 near ";"; process sensitivity list is empty
[22]     end block;
[23] end;
-->[fatal] line 23 near ";"; interface name I2 not found
[24]
Number of errors detected: 1
Errors in VHDL source. No VIA produced
VHDL Analysis complete.
```

tested portions of VHDL that were not implemented were counted and removed from the test suite. Second, those tests that tested both implemented and non-implemented language features were specially handled. These tests were split into two tests--one tested implemented features and became part of the test suite, and the other tested the non-implemented features and was removed.

The results of each run of the test suite are summarized below in Table 5.1. Because the Analyzer does not semantically analyze the entire VHDL language, it will sometimes detect errors in otherwise correct VHDL source. Other messages showed where the Analyzer ignored parts of the language that were not implemented in any manner. The error messages tell the user what restrictions the Analyzer is using and what actions the Analyzer took.

Error testing. Error tests using the VHDL VMS test suite were run on the final release of the Analyzer. The results of error testing were given above. The higher percentage of passed tests resulted because many of the tests involved erroneous input that was not syntactically correct. These errors were caught by the parser generated automatically by yacc from the VHDL grammar. Other error tests were constructed based on knowledge of the internal structure of the Analyzer and were designed to catch implementation errors³. These were not part of the original VHDL VMS Analyzer test suite.

Portability Testing

Using the conformance and error tests discussed above, the Analyzer was tested on three different computer systems, the VAX 11/780, the ELXSI 6400, and the Sun 2 Workstation. Simple VHDL designs, discussed in Chapter 1, were tested, as

³ This is known as *white-box* (or *glass-box*) testing, because it assumes that one knows how the program being tested operates.

Table 5.1 Results of VMS Test Suite Testing

Result	Short	Error	Total
Pass	230 (68%)	378 (81%)	608 (76%)
Fail	42 (13%)	72 (15%)	114 (14%)
Not Impl	65 (19%)	19 (4%)	84 (10%)
Total	337	469	806

were the performance tests discussed later in this chapter. Portability testing showed whether the Analyzer had system-specific code that would prevent others from using this research. For example, if the Analyzer used a C function to obtain information from the operating system (such as the date or the amount of memory on the processor), then this (or an equivalent) function must be available on other processors. Because the Analyzer was written with no such system-specific code, there were no portability problems.

Performance Testing

Speed. Analyzer execution speed was measured by averaging the compile times for several VHDL designs which used several different types of design methods, so as not to be biased in favor or against the implementation of the Analyzer. "Wall-clock" time and CPU times were measured with the UNIX *time* command (Bell, 1983) and the results are presented in Table 5.2, along with the times for the VHDL/VMS Analyzer. The Analyzer's performance is well within the requirements of 1000 lines per CPU minute. In

Table 5.2 Results of Portability Tests

Design Type	Processor			
	VAX VMS	VAX UNIX	SUN	ELXSI
Dataflow1	32.92 ¹ (0.69) ²	2.63 (0.05)	1.39 (0.03)	0.91 (0.21)
Procedural	33.66 (1.71)	2.52 (0.05)	1.34 (0.07)	0.90 (0.22)
Dataflow2	----- (---)	2.22 (0.08)	1.13 (0.07)	0.85 (0.20)
¹ Time in CPU seconds. ² standard deviation				

fairness, it must be noted that since the VHDL/VMS Analyzer is written in Ada and is not a subset analyzer, its analysis times were expected to be slower. This is because of the run-time checks that Ada performs that C (the implementation language of the UNIX Analyzer) does not. Also, the full VMS version performs more semantic checks on the input than the UNIX subset Analyzer. Finally, the VHDL/VMS Analyzer uses a design library which increases the system overhead due to the opening and closing of design library files. Therefore, it can be conjectured that because the UNIX VHDL Analyzer uses C and does not use a design library, a full implementation will still be somewhat faster than the VMS version.

Memory Usage. For each of the performance tests described above, the memory usage was measured (again using the *time* command). This usage varied from 300K to over 800K. Memory usage depends on the number of VHDL objects

defined and not on the number of VHDL statements. This is because the symbol table is dynamically created and the operation table is a static array.

Disk Usage. The goal for this project (Chapter 1) was 100 bytes of VIA for each VHDL line. This criterion was tested by measuring the size of the resulting VIA files from the analysis of the aforementioned VHDL designs. The average size of each design was 51 lines and, therefore, the goal was a VIA file of 5100 bytes or less in size. The average VIA file size was 5180 bytes, which was only slightly above the goal for this project.

Integration Testing

Integration tests were the culmination of the Analyzer test phase. Their purpose was to ensure that the interface between the Analyzer and an AVE tool (the UNIX VHDL Simulator (Kodama, 1987)) was operable and provided the information necessary to process a VHDL design.

Method. Using the VHDL benchmarks developed by Dukes (Dukes, 1987), each design was analyzed and the resulting VIA submitted to the simulator along with a set of test vectors. To verify that the design was correctly analyzed and simulated, the outputs from the Analyzer and the simulator were compared to the same simulation performed using the VMS VHDL environment and to a manual simulation (of the simpler designs).

Results and Comparisons with VMS VHDL. The results for the Analyzer portion of the integration tests have been given previously in Table 5.2. For a comparison of the VMS VHDL Simulator and its UNIX counterpart, see (Kodama, 1987).

Chapter Summary

This chapter has showed the manner in which the Analyzer was tested and the results of those tests. Recommendations for future research with the Analyzer and final conclusions are presented in the final chapter.

VI. Conclusions and Recommendations

This thesis has presented the development and implementation of a UNIX-based VHDL analyzer. Future research, recommended below, will further expand its capabilities. The direction this research takes, however, will depend on the conclusions presented first in the following section.

Conclusions

Selection of VIA. The effort in developing a new VIA has been well rewarded. After its design was finished, work on an AVE UNIX simulator has been able to proceed rapidly and with visible results. The subset of VHDL that was implemented in this project proved robust enough to allow the design and simulation of several useful VHDL circuit descriptions. A subgoal of this research--to demonstrate "end-to-end" throughput of a VHDL design--was attained. As was mentioned before, several portions of the VHDL language were not implemented. These portions (Appendix A) include separate compilation, configurations, revision specifications, user attributes, and floating point arithmetic, as well as several of the semantic checks on those constructs that were implemented. The elimination of these capabilities from the analyzer did not detract from its

usefulness: using the analyzer and simulator, the beginnings of a VHDL circuit library have taken shape.

Test Results. Based on the tests described in Chapter 5, the major goals of this project have been successfully completed:

1. The Analyzer runs under UNIX on several processors.
2. The Analyzer has been subjected to thorough testing.
3. The Analyzer produces an intermediate representation that meets the design requirements for compactness and efficiency.
4. The Analyzer correctly analyzes more than 75% of VHDL Version 7.2. As was mentioned in the preceding section, this subset is robust enough to describe many of the VHDL designs that students would be generating.
5. The Analyzer generally performs within its performance goals. Although memory usage was larger than expected, many UNIX environments have up to 16 Megabytes of main memory. The Analyzer's usage of memory was, therefore, deemed acceptable.
6. The Analyzer has been tested in conjunction with the UNIX VHDL Simulator.

Thus, this implementation can be described as production-quality based on the above evaluations¹. More work can be done to improve its conformance and error handling qualities, as well as its use of main memory.

¹ Source code for the Analyzer is available from the Department of Mathematics and Computer Science, Air Force Institute of Technology, Wright-Patterson AFB, OH 45433.

Recommendations for Future Research

At least five topics for further research resulted from this project.

Implement Entire Language. It is envisioned that the analyzer, at some future date, will be able to generate VIA for the entire language. Many of the design decisions during this project were made with this extension in mind. For example, the structure of the symbol table was designed to support separate compilation. While the analyzer now rounds floating point numbers to the nearest integer, the future introduction of real numbers is facilitated by separation of integer and floating point literals in both the grammar and the supporting routines (even though both are currently mapped to *long integers*). At a later time, code could be added to the supporting routines to implement floating point without affecting the code for integers.

Add to User Options. The only user option implemented in the analyzer was a "verbose" option that generates a listing file (with error messages, if any); other options were not necessary during the analyzer's development. But, as the analyzer matures, it should offer the user the following options:

1. Build option. After separate compilation is implemented, the analyzer should have the capability of either creating separate VIA files for each design unit in a

design file or a completely linked VIA file that could be used directly by the simulator.

2. Context option. The user should have the capability of instructing the analyzer where the VIA files for the separate components of his/her design can be found. This would allow the creation of a central VHDL library for commonly used components or several libraries, each having its designs implemented using one of VHDL's three design implementations.

3. Generate option. This option allows the user to direct the creation of VIA files to any subdirectory.

Optimize VIA. In several areas the VIA could be optimized. First, when generating a complete VIA file from separate modules, only those objects in the VIA symbol table that are referenced need be output to disk. This method would, over the course of time, save much disk storage space since many of the objects in user defined packages and in the pre-defined package STANDARD (Intermetrics, 1985b: Appendix A) are not referenced in every VHDL design. Second, code improving optimizations could be applied to the operation table of VIA. These optimizations include common subexpression elimination, copy propagation, and the detection of loop-invariant computations (Aho, Sethi, and Ullman, 1986:Ch. 10). Finally, even though storing the VIA as a binary file, rather than in ASCII, could save disk space, I suggest the VIA remain as an ASCII file. The use of ASCII

files was necessary during the design phases of both the analyzer and the simulator, so that the output of the analyzer could be visually checked. If there was an error with the analyzer, the VIA could be manually corrected with a text editor before being used by the simulator. This use of ASCII files allowed the simulator development to proceed without having to wait for every error to be removed from the analyzer. Now that the initial development phases are completed, the usefulness of an ASCII VIA file continues. An ASCII VIA file can be used with UNIX source code management utilities such as the Source Code Control System (SCCS) (Rochkind, 1985) and the Revision Control System (RCS) (Tichy, 1982) (which do not handle binary files). These utilities could be used to implement *revisions* of design units, where the user specifies which version of the design unit should be used and the utility retrieves that version of VIA from the revision file and writes it to disk. This method would also save considerable disk space since only the base version of a VIA representation is saved in its entirety. As other versions are added, only the differences needed to create those versions are saved. This philosophy of using existing UNIX tools follows from the requirements of this project.

Add Design Library. As more persons begin using the Analyzer for more complicated designs, its memory usage could become a problem. A design library could alleviate this problem by supplying symbol table information so that

the analyzer would not have to load an entire VIA file to access one definition. The form of the design library could range from several random VIA access routines to an integrated database manager. Such a database manager could possibly be built with artificial intelligence techniques to detect inconsistencies in VHDL designs introduced through modification of one of the design components. It could also help the user design and select the components of his/her circuit.

Implement other AVE Tools. Other tools to facilitate UNIX VHDL design are needed. Among them are *reverse analyzers* and *silicon compilers*. Reverse analyzers were discussed earlier. Going the other way, a silicon compiler would read a VIA representation and help to generate a cell layout for the circuit that could be manufactured into an actual chip. With both of these tools available, a designer could write a VHDL description of a circuit, generate the corresponding VIA, use the VIA to generate the cell layout, manufacture the chip, and then reverse engineer the chip (using the reverse analyzer) to obtain the VHDL description of the manufactured chip. Then the original VHDL description could be compared to the VHDL that resulted from the chip analysis to detect possible manufacturing errors.

Summary

The UNIX VHDL analyzer produced during this research provides the academic community with a necessary tool for

the acquisition of further knowledge of VHSIC design. The need for such tools becomes more and more evident as new technology increases the complexity of electronic circuits. A standardized manner to describe such circuits can be found in the use of VHDL. Through ongoing research, such as the project presented in this paper, the availability of VHDL can be broadened for the mutual benefit of industry and academia, which, in turn, directly benefits the Air Force.

AD-A188 832

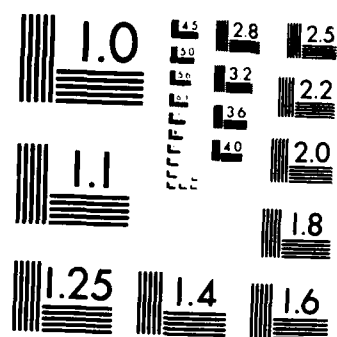
A PRODUCTION-QUALITY UNIX VERY HIGH SPEED INTEGRATED
CIRCUIT (VHSIC) HARD. (U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. R M BRATTON
DEC 87 AFIT/GCS/HA/87D-1 F/G 12/5

2/2

UNCLASSIFIED

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

APPENDIX A. AFIT VHDL ANALYZER IMPLEMENTATION

This section outlines the status of the AFIT VHDL Analyzer at the time of the completion of this report. It, like the similar appendix in Frauenfelder's thesis, is keyed to the VHDL Language Reference Manual, by chapter. If a particular function is listed as "not implemented" then the analyzer will only check the syntax of the construct and no VIA is generated.

Chapter 1: Design Entities

Implemented except for *next level configuration*.

Chapter 2: Subprograms

Implemented.

Chapter 3: Packages

Implemented.

Chapter 4: Types

Multidimensional array types not implemented.

All predefined types implemented except *character names*.

Floating point types mapped to integer values.

Chapter 5: Declarations

Alias declarations not implemented.

User-defined attributes not implemented.

Chapter 6: Specifications and Directives

Attribute specification not implemented.

Select directive not implemented.

Entity aspect, port and generic map aspects, and body aspect not implemented.

Configuration specification and binding indication not implemented.

Chapter 7: Names and Expressions

Names of statement labels are ignored when used as *selected names*.

Library names not implemented.

Context and revision specifications not implemented.

Indexed names of more than one dimension are not implemented.

User-defined attribute names are not implemented

Mod and *rem* implemented, but actual values not calculated for constants.

Chapter 8: Statements

Exit statement not implemented.

Procedure call statement not implemented.

Generate statement not implemented.

Chapter 9: Scope and Visibility

Objects cannot be selected through use of a statement label.

Overloading not implemented.

Chapter 10: Design Units and Their Analysis

Separate compilation not implemented.

Appendix A: Lexical elements

Alternate replacement characters not implemented.

Maximum identifier length is 255 characters.

Appendix B: Predefined Language Environment

Only predefined *attributes* for scalar types implemented.

Character names (NULL .. DEL) not implemented.

Time functions TMIN and TMAX not implemented.

APPENDIX B. VIA DEFINITION

The following is the format for the VIA intermediate file.

File Definition: The VIA file is an ASCII file consisting of 4 major divisions: header, symbol table, operation table, and string table. The header provides information to enable the access of the other tables.

BNF: In this definition, nonterminals are surrounded by angle brackets "< >". Comments are preceded by two hyphens, as in VHDL and Ada.

```
<via_file> ::= <header> <syntab> <optab> <strtab>
<header> ::= <#_design_units> <#_op_recs> <#_sym_recs> <#_chars>
<#_design_units> ::= integer    -- number of design units
<#_op_recs> ::= integer        -- number of entries in the operation table
<#_sym_recs> ::= integer        -- number of entries in the symbol table
<#_chars> ::= integer           -- number of characters in the string table
<syntab> ::= <sym_rec> | <sym_rec> |
<sym_rec> ::= <unit> | <object> | <type>
<unit> ::= <interf> | <arch> | <config> | <pack> | <proc> | <func>
```



```

<interf> ::= <rec_num> <sym_name> UNIT INTERFACE <info> <vis>
            <sym_ref>  -- record number of first port
            <sym_ref>  -- record number of first generic
            <sym_ref>  -- record number of first declaration
            <op_ref>   -- record number of first directive
            <sym_next> -- record number of next declaration

<arch> ::= <rec_num> <sym_name> UNIT ARCHITECTURE <info> <vis>
            <sym_ref>  -- record number of interface
            <op_ref>   -- record number of block statement
            <sym_next> -- record number of next declaration

<config> ::= <rec_num> <sym_name> UNIT CONFIGURATION <info> <vis>
            <sym_ref>  -- record number of interface
            <sym_ref>  -- record number of architecture body
            <op_ref>   -- record number of next level configuration
            <sym_next> -- record number of next declaration

<pack> ::= <rec_num> <sym_name> UNIT PACKAGE <info> <vis>
            <sym_ref>  -- record number of first package declaration
            <sym_next> -- record number of next declaration

<proc> ::= <rec_num> <sym_name> UNIT PROCEDURE <info> <vis>
            <sym_ref>  -- record number of first parameter
            <op_ref>   -- record number of first statement
            <sym_next> -- record number of next declaration

<func> ::= <rec_num> <sym_name> UNIT FUNCTION <info> <vis>
            <sym_ref>  -- record number of first parameter
            <sym_ref>  -- record number of return type
            <op_ref>   -- record number of first statement
            <sym_next> -- record number of next declaration

<object> ::= <sig> | <var> | <const> | <label> | <elem> | <param> |
            <alias> | <comp>

<sig> ::= <rec_num> <sym_name> OBJECT SIGNAL <info> <vis>
            <sym_ref>  -- record number of type
            <sym_ref>  -- record number of resolution function
            <op_ref>   -- record number of initialization expression
            <sym_next> -- record number of next declaration

<var> ::= <rec_num> <sym_name> OBJECT VARIABLE <info> <vis>
            <sym_ref>  -- record number of type
            <sym_ref>  -- record number of resolution function
            <op_ref>   -- record number of initialization expression
            <sym_next> -- record number of next declaration

```

```

<const> ::= <rec_num> <sym_name> OBJECT CONSTANT <info> <vis>
           <sym_ref>      -- record number of type
           <sym_ref>      -- record number of resolution function
           <op_ref>       -- record number of initialization expression
           <val>          -- constant value
           <sym_next>     -- record number of next declaration

<label> ::= <rec_num> <sym_name> OBJECT LABEL <info> <vis>
           <op_ref>       -- record number of first statement
           <op_ref>       -- record number of initialization expression
           <sym_next>     -- record number of next declaration

<elem> ::= <rec_num> <sym_name> OBJECT ELEMENT <info> <vis>
           <sym_ref>      -- record number of type
           <sym_ref>      -- record number of record type
           <sym_next>     -- record number of next declaration

<param> ::= <rec_num> <sym_name> OBJECT PARAMETER <info> <vis>
           <sym_ref>      -- record number of type
           <sym_ref>      -- record number of subprogram
           <op_ref>       -- record number of initialization expression
           <sym_next>     -- record number of next declaration

<alias> ::= <rec_num> <sym_name> OBJECT ALIAS <info> <vis>
           <sym_ref>      -- record number of base type
           <sym_ref>      -- record number of aliased object
           <sym_next>     -- record number of next declaration

<comp> ::= <rec_num> <sym_name> OBJECT COMPONENT <info> <vis>
           <sym_ref>      -- record number of first port
           <sym_ref>      -- record number of first generic
           <sym_next>     -- record number of next declaration

<type> ::= <irange> | <rrange> | <enum> | <array> | <record> |
           <physical> | <subtype>

<irange> ::= <rec_num> <sym_name> TYPE IRANGE <info> <vis>
            <sym_ref>      -- record number of base type
            integer       -- minimum range
            integer       -- maximum range
            <sym_next>    -- record number of next declaration

<rrange> ::= <rec_num> <sym_name> TYPE RRANGE <info> <vis>
            <sym_ref>      -- record number of base type
            real          -- min range
            real          -- max range
            <sym_next>    -- record number of next declaration

```

```

<enum> ::= <rec_num> <sym_name> TYPE ENUM <info> <vis>
        integer      -- number of literals
        integer      -- value of first (always 0)
        integer      -- value of last
        <sym_ref>    -- record number of first literal
        <sym_next>   -- record number of next declaration

<array> ::= <rec_num> <sym_name> TYPE ARRAY <info> <vis>
        <sym_ref>    -- record number of index type
        <sym_ref>    -- record number of element type
        integer      -- value of lower bound
        integer      -- value of upper bound
        <sym_next>   -- record number of next declaration

<record> ::= <rec_num> <sym_name> TYPE RECORD <info> <vis>
        <sym_ref>    -- record number of first field
        <sym_next>   -- record number of next declaration

<physical> ::= <rec_num> <sym_name> TYPE PHYSICAL <info> <vis>
        <sym_ref>    -- record number of base unit
        integer      -- number of units
        integer      -- minimum value
        integer      -- maximum value
        <sym_next>   -- record number of next declaration

<subtype> ::= <rec_num> <sym_name> TYPE SUBTYPE <info> <vis>
        <sym_ref>    -- record number of base type
        <sym_ref>    -- record number of resolution function
        integer      -- minimum range
        integer      -- maximum range
        <sym_next>   -- record number of next declaration

<sym_name> ::= character string

<info> ::= <info_field> | <info> <info_field>

<info_field> ::= NO_INFO | PORT | GENERIC | PARAM | MODE_IN |
        MODE_OUT | MODE_IO | MODE_BUF | MODE_LINK | ATOMIC |
        GRANULAR | STATIC | LOOP_VAR | ARCH_CHILDLESS |
        ARCH_STRUCTURE

<vis> ::= integer      -- scope of definition's visibility

<sym_ref> ::= integer  -- record number of SYMTAB record

<op_ref> ::= integer   -- record number of OPTAB record

<sym_next> ::= integer -- record number of next SYMTAB definition

<optab> ::= <op_rec> | <op_rec> |

```

```

<op_rec> ::= <rec_num> <op_name> <line_num> <label> <type>
           <first> <op_next> <value>

<rec_num> ::= integer    -- unique record number

<op_name> ::= ABS | ADD | AFTER | ALT_CLAUSE | AND | ASSERT |
             ASSOC | ATTRIBUTE | BINDING | BLOCK | CASE |
             CHOICE_CLAUSE | COMPONENT | COND_CLAUSE | CONST_LIT |
             CONST_STR | DISABLE | DIV | E_VALUE | ELMT_OF |
             ENABLE | EQ | EXP | FOR | FUNC_CALL | GE | GENERIC |
             GT | GUARD | I_VALUE | IF | INITIALIZE | LE | LOOP |
             LT | MOD | MUL | NAND | NE | NEXT | NOOPR | NOR | NOT |
             NULL | NULLBR | OPEN | OR | OTHERS | PORT | PROC_CALL |
             PROCESS | R_VALUE | RANGE | REM | REPORT | RETURN |
             SENS_LIST | SEVERITY | SIG_ASSIGN | SUB | SYM_REF |
             TRANSPORT | VAR_ASSIGN | VOID_NODE | WAVE | WHEN |
             WHILE | XOR

<line_num> ::= integer    -- line number from source VHDL

<label> ::= <sym_ref>     -- symbol table index to label for this
                       -- statement

<type> ::= <sym_ref>      -- symbol table index for the type of this
                       -- statement

<first> ::= <op_ref>      -- operation table index for first pointer

<op_next> ::= <op_ref>    -- operation table index for next pointer

<val> ::= integer | real  -- value for constant literals

<strtab> ::= { <string> }

<string> ::= character | character | '\0'

```

Appendix C. VHDL ANALYZER TEST SUITE

This appendix details the number of test objectives (Intermetrics, 1984a:19-20) and tests for each chapter of the Language Reference Manual. These tests have been formulated to check for each requirement in the LRM and are similar to the examples shown in Chapter 5. However, due to copyright restrictions, the actual tests in the VHDL Analyzer Test Suite cannot be published in this thesis.

Chapter 1: Design Entities (58 objectives)

Short tests: 13 Error tests: 22

Chapter 2: Subprograms (20 objectives)

Short tests: 7 Error tests: 14

Chapter 3: Packages (4 objectives)

Short tests: 2 Error tests: 3

Chapter 4: Types (82 objectives)

Short tests: 37 Error tests: 41

Chapter 5: Declarations (51 objectives)

Short tests: 36 Error tests: 43

Chapter 6: Specifications and Directives (150 objectives)

Short tests: 26 Error tests: 54

Chapter 7: Names and Expressions (59 objectives)

Short tests: 71 Error tests: 84

Chapter 8: Statements (66 objectives)

Short tests: 68 Error tests: 107

Chapter 9: Scope and Visibility (14 objectives)

Short tests: 21 Error tests: 21

Chapter 10: Design Units and Their Analysis (11 objectives)

Short tests: 14 Error tests: 21

Appendix A: Lexical elements (47 objectives)

Short tests: 0 Error tests: 34

Appendix B: Predefined Environment

Short tests: 42 Error tests: 24

Bibliography

- Aeronautical Systems Division, Air Force Systems Command.
VHSIC Hardware Description Language (VHDL) Program.
Solicitation No. F33615-83-R-1003. Wright-Patterson
AFB OH, 30 Mar 1983.
- Aho, Alfred V. and S.C. Johnson. "LR Parsing." Computing Surveys, 6 (2): 99-124 (June 1974).
- Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman.
Compilers: Principles, Techniques, and Tools.
Reading, MA: Addison-Wesley Publishing Co., 1986.
- Aylor, J.H., R. Waxman, and C. Scarratt. "VHDL--Feature Description and Analysis." IEEE Design and Test of Computers, 3: 17-27 (April 1986).
- Barrett, William A. and John D. Couch. Compiler Construction: Theory and Practice. Chicago: Science Research Associates, Inc., 1979.
- Bell Telephone Labs, Inc. UNIX Programmer's Manual, 4.2 BSD. Berkely, CA: August 1983.
- Brooks, Frederick P., Jr. The Mythical Man-Month. Reading MA: Addison-Wesley Publishing Co. 1975.
- Carter, Lt Col Harold W. and others. 1986 Research Report: AFIT VHDL/DB/DBMS Research. AFIT-ENC-TR-87-01. Air Force Institute of Technology (AU), Wright-Patterson AFB OH, January 1987.
- CAD Language Systems, Inc. "VHDL Language Reference Manual (Draft Standard 1076/A)," CAD Language Systems, Inc., Rockville, MD, 31 December 1986.
- Chu, Y. Computer Organization and Microprogramming. Englewood Cliffs NJ: Prentice-Hall, 1972.
- Conway, R.W. and T.R. Wilcox. "Design and Implementation of a Diagnostic Compiler for PL/I," Communications of the ACM, 16 (3): 169-179 (March 1973).
- Dallen, John Anthony, Jr. The Synthesis and Validation of Experimental VLSI Design Using Decoupled Behavioral, Structural and Physical Specifications. PhD Dissertation. Department of Computer Science, Duke University, Durham NC, 1983.

- . Patois User's Manual. Department of Computer Science, Duke University, Durham, NC, 1983b.
- Decker, Capt Sharon L. A Study and Implementation of An Automatically Retargeting Microcode Compiler System. MS Thesis GCS/ENG/86D-9. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986.
- Deitel, Harvey M. An Introduction to Operating Systems. Reading MA: Addison-Wesley Publishing Company, 1984.
- Dewey, Alan and Anthony Gadiant. "VHDL Motivation," IEEE Design & Test of Computers, 3 (2): 12-16 (April 1986).
- Dukes, CPT Michael A. Examination of a VHDL Simulator on SISD and MIMD Systems. Technical Report. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, to be published.
- Evans, Arthur, Kenneth J. Butler, G. Goos, and William A. Wulf. DIANA Reference Manual Revision 3. Tartan Laboratories, Inc. Contract No. MDA903-82-C-0148. Pittsburgh PA, 28 February 1983.
- Frauenfelder, Capt Deborah J. An Implementation of a Language Analyzer for the Very High Speed Integrated Circuit Hardware Description Language. MS Thesis GCE/MA/86D-1. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986.
- . Telephone interview. Colorado Springs, CO, 29 June 1987.
- Gilman, Alfred S. "VHDL--The Designer Environment," IEEE Design & Test of Computers, 3 (2): 42-47 (April 1986).
- Goodenough, John B. The Ada Compiler Validation Capability Implementer's Guide (Version 1). Wright-Patterson AFB OH: SofTech Inc., December 1986.
- Goodman, Glenn W. "VHSIC a Potential Boon to Air Force R&M and Advanced Tactical Fighter," Armed Forces Journal International, 124 (5): 58-64 (January 1987).
- Holt, R.C. and D.T. Barnard. "Syntax-Directed Error Repair and Paragraphing," Computer Systems Research Group, University of Toronto, 1976.

- Horning, J.J. "What The Compiler Should Tell the User,"
Compiler Construction: An Advanced Course (Second
Edition), edited by F.L. Bauer and J. Eickel. New
York: Springer-Verlag, 1976.
- Intermetrics, Inc. VHDL Analyzer Program Specification.
Contract F33615-83-C-1003. Bethesda MD, 30 July 1984a.
- . VHDL Analyzer Test Plan. Contract F33615-83-C-1003.
Bethesda MD, 30 July 1984b.
- . VHDL Design Library Specification. Contract F33615-
83-C-1003. Bethesda MD, 1 August 1985a.
- . VHDL Language Reference Manual Version 7.2.
Contract F33615-83-C-1003. Bethesda MD, 1 August
1985b.
- . VHDL User's Manual: Volume I - Tutorial. Contract
F33615-83-C-1003. Bethesda MD, 1 August 1985c.
- . VHDL User's Manual: Volume II - User's Reference
Guide. Contract F33615-83-C-1003. Bethesda MD, 1
August 1985d.
- Johnson, S.C. "Yacc: Yet Another Compiler-Compiler,"
Murray Hill, NJ: AT&T Bell Laboratories, 31 July 1978.
- Kodama, 2nd Lt Harvey. The UNIX VHDL Simulator. MS Thesis,
School of Engineering, Air Force Institute of
Technology (AU), Wright-Patterson AFB OH, December
1987.
- Kamrowski, Capt M.S. Design of a Parallel Simulator for the
VHDL. MS Thesis. School of Engineering, Air Force
Institute of Technology (AU), Wright-Patterson AFB OH,
December 1986.
- Kantorowitz, E. and H. Laor. "Automatic Generation of
Useful Syntax Error Messages," Software--Practice and
Experience, 16 (7): 627-640 (July 1986).
- Kernighan, Brian W. and Dennis M. Ritchie. The C
Programming Language. Englewood Cliffs NJ: Prentice-
Hall, Inc., 1978.
- Knapp, David W. and Alice C. Parker. A Data Structure for
VLSI Synthesis and Verification. Contract DAAG29-80-k-
0083. Department of Electrical Engineering-Systems,
University of Southern California, Los Angeles CA, 8
May 1984.

- Lesk, M.E. and E. Schmidt. "Lex--A Lexical Analyzer Generator," Murray Hill, NJ: Bell Laboratories, 31 July 1978.
- Levitan, Steven P. Architectural Simulation of Digital Systems. Project proposal. Department of Electrical Engineering, University of Pittsburgh, Pittsburgh PA, 22 April 1987.
- Lieberherr, Karl J. "Toward a Standard Hardware Description Language," IEEE Design and Test of Computer of Computers, 2 55-62 (February 1985).
- Lipsett, Roger, Erich Marschner, and Moe Shahdad. "VHDL--The Language," IEEE Design and Test of Computers, 3 (2) 28-41 (April 1986).
- Lynch, Major W.L. A Kernel VHDL Simulator. MS Thesis. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986.
- Morgan, H.L. "Spelling Correction in Systems Programs," Communications of the ACM, 13 (2): 90-94 (February 1970).
- Nash, J.D. "Bibliography of Hardware Description Languages," ACM SIGDA Newsletter, 14 (1) 18-37 (February 1984).
- Nash, J.D. and Larry F. Saunders. "VHDL Critique," IEEE Design & Test of Computers, 3 (2): 54-65 (April 1986).
- Noonan, Robert E. "An Algorithm for Generating Abstract Syntax Trees," Computer Language, 10 (3/4): 225-236 (1985).
- Piloty, Robert, and Dominique Borriane. "The Conlan Project: Concepts, Implementations, and Applications," Computer 81-82 (February 1985).
- Pressman, Roger S. Software Engineering: A Practitioner's Approach. New York: McGraw-Hill Book Company, 1982.
- Rochkind, M.J. "The Source Code Control System," IEEE Transactions on Software Engineering Vol. SE-1: 4 (Dec 1985).
- Schreiner, A.V., and H.G. Friedman, Jr. Introduction to Compiler Construction with UNIX. Englewood Cliffs: Prentice-Hall, Inc., 1985.

- Tichy, W.F. "Design, Implementation, and Evaluation of a Revision Control System," Proceedings of the 6th International Conference on Software Engineering (Sep 1982).
- Tremblay, Jean-Paul and Paul G. Sorenson. The Theory and Practice of Compiler Writing. New York: McGraw-Hill Book Company, 1985.
- Walker, Robert A. and Donald E. Thomas. "A Model of Design Representation and Synthesis," Proceedings of the 22nd Design Automation Conference 1985: Paper 28.1, 453-459.
- Waxman, Ron. "Hardware Design Languages for Computer Design and Test," Computer, 19 (4): 90-97 (April 1986).
- Wichmann, Brian A. and Z.J. Ciechanowicz. Pascal Compiler Validation. Chichester Great Britain: John Wiley & Sons, 1983.

VITA

Captain Randolph M. Bratton was born on 11 October 1949 in Atlanta, Georgia and was raised in Greensboro, North Carolina. He graduated from Guilford High School in 1967 and attended North Carolina State University from 1967 until 1972, majoring in Aerospace Engineering and Liberal Arts-Spanish. In 1973, he enlisted in the United States Navy as a Musician Seaman and served until his honorable discharge in 1980. He was stationed aboard the *USS Forrestal* during her 1974 Mediterranean cruise and served with the Navy bands at Orlando, Florida and Memphis, Tennessee. He completed work on the degree of Bachelor of Science from the University of the State of New York (External Degree Program) in September 1979. In January 1981, he entered Memphis State University, pursuing a second Bachelor degree in Computer Technology as well as a commission from the USAF through the ROTC program. After graduation in December 1982, he was selected as an AFROTC Distinguished Graduate and received a Regular commission. After his return to active duty in January 1983, he served as a computer systems analyst at the Aeronautical Systems Division, Wright-Patterson AFB, Ohio until entering the School of Engineering, Air Force Institute of Technology, in June 1986.

Permanent Address: 204 Brushwood Dr.

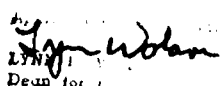
Greensboro, North Carolina 27410

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE


REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified			1b. RESTRICTIVE MARKINGS A188 832		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.		
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/MA/87D-1			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION School of Engineering		6b. OFFICE SYMBOL (If applicable) AFIT/ENC		7a. NAME OF MONITORING ORGANIZATION	
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology (AU) Wright-Patterson AFB, Ohio 45433-6583			7b. ADDRESS (City, State, and ZIP Code)		
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Air Force Wright Aeronautical Labs		8b. OFFICE SYMBOL (If applicable) AFWAL/AADE		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB, Ohio 45433			10. SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.
			WORK UNIT ACCESSION NO.		
11. TITLE (Include Security Classification) See box 19.					
12. PERSONAL AUTHOR(S) Randolph M. Bratton, B.S., B.S.E.T., Captain, USAF					
13a. TYPE OF REPORT MS Thesis		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Year, Month, Day) 1987 December	
				15. PAGE COUNT 113	
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP			
I2	05		VHDL Compilers		
			VHSIC Hardware Description Language		
			VIA UNIX		
19. ABSTRACT (Continue on reverse if necessary and identify by block number)					
Title: A Production-Quality UNIX Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) Subset Analyzer					
Thesis Chairman: Richard R. Gross, Lt. Col., USAF					
<div style="text-align: right;">  Lynn J. Linn Dean for Research and Technical Development Air Force Institute of Technology (AFIT) Wright-Patterson AFB OH 45433 </div>					
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL Richard R. Gross, LtCol, USAF			22b. TELEPHONE (Include Area Code) (513) 255-3098		22c. OFFICE SYMBOL AFIT/ENC

Abstract

This paper describes the design and implementation of the Air Force Institute of Technology's (AFIT's) UNIX-based VHDL Analyzer. The purpose of this tool is to facilitate the introduction of VHDL into the academic environment, which may not be able to use the Department of Defense's VMS-based software. This research emphasized two areas: the criteria for a "production-quality" software product and the design of an efficient Intermediate Representation (IR) that serves as an interface between the Analyzer and other tools in the AFIT VHDL Environment (AVE). Background on other UNIX VHDL analyzers, as well as other IRs, was presented. A two-part IR, based on Dallen's *Patois* hardware description language and named the VHDL Intermediate Access (VIA), was designed, and examples were given that illustrate its use. Test results showed that the Analyzer passed over 75% of the conformance tests from the VHDL VMS Analyzer Test Suite and performed well in the areas of compile time, memory usage, and disk usage. Recommendations for future research include adding user options to the Analyzer and implementing a design library for VHDL designs.



END

FILMED

MARCH, 19 88

DTIC